ENERGYPLUS™

Guide for Module Developers

Everything You Need to Know about EnergyPlus Calculational Development

(but were hesitant to ask)

Date: April 26, 2002

TABLE OF CONTENTS

Introduction 1

Modules in EnergyPlus 2

What is a module anyway? 2

Program Modules 2

Data Only Modules 2

What is a module developer? 2

Input Concepts 3

Input Data Dictionary 3

Input Data File 4

Input Considerations 5

Module Structure 8

Module Outline 8

Module Example 10

How it fits together 26

Top Level Calling Tree 26

High Level HVAC Calling Tree 26

Air System Calling Tree 27

Plant Supply Calling Tree 27

Zone Equipment Calling Tree 28

Inserting the New Module into the Program 29

Considerations for Legacy Codes 31

Code Readability vs. Speed of Execution 32

EnergyPlus Services 33

Global Data 33

Input Services 33

GetNumObjectsFound 33

TABLE OF CONTENTS

GetObjectItem 33

GetNodeNums 34

GetObjectItemNum 35

FindItemInList 35

SameString 35

VerifyName 35

Schedule Services 35

GetScheduleIndex 38

GetCurrentScheduleValue 38

Performance Curve Services 38

GetCurveIndex 38

CurveValue 39

Fluid Property Services 39

Using Fluid Property Routines in EnergyPlus Modules 39

Table 1. Regions for Fluid Properties 40

Fluid Property Data and Expanding the Refrigerants Available to EnergyPlus 40

Other Useful Utilities 44

GetNewUnitNumber 44

FindUnitNumber 44

Weather Services 44

Error Messages 44

Flags and Parameters 45

Parameters 45

Simulation Flags 45

Psychrometric services 48

rhoairfn(pb,tdb,w) 48

TABLE OF CONTENTS

cpairfn(w,T) 48

PSYDPT(TDB,TWB,PB) 48

PSYDPW(W,PB) 48

PSYHTW(TDB,W) 48

PSYHTR(TDB,RH,PB) 49

PSYTHW(H,W) 49

PSYRHT(TDB,W,PB) 49

PSYTWD(TDB,W,PB) 49

PSYVTW(TDB,W,PB) 49

PSYWDP(TDP,PB) 49

PSYWTH(TDB,H) 49

PSYWTP(TDB,TWB,PB) 49

PSYWTR(TDB,RH,PB) 49

SATUPT(T) 49

SATUTH(H,PB) 50

SATUTP(P) 50

HVAC Network 51

Branches, Connectors, and Nodes 51

Figure 1. HVAC Input Diagram 52

Nodes in the simulation 53

Data Flow in an HVAC Component Module 54

Node Mass Flow Variables 57

Output 59

How Do I Output My Variables? 59

Table 2. SetupOutputVariable Arguments 60

Output Variable Dos and Don’ts 61

TABLE OF CONTENTS

What Variables Should I Output? 61

Output Variable Naming Conventions 61

What are Meters? 61

How Do I Create A Meter? 62

Rules for Meter Variables 62

Important Rules for Module Developers 64

Appendix A. DataGlobals Module 65

Appendix B. DataEnvironment Module 69

Introduction

EnergyPlus is a modular simulation program designed to model the performance, energy consumption and pollutant production of a building. EnergyPlus models energy transport through the building envelope, heat gains within the building, and all the HVAC equipment used to heat and cool the building. The program is designed for ease of development. The concept is that many people will contribute to EnergyPlus and the program structure has been designed to make this possible.

EnergyPlus is written entirely in Fortran 90. Fortran 90 is a powerful modern programming language with many features. Using Fortran 90 it is possible to program in many different styles. The EnergyPlus team has chosen a particular style that emphasizes code extensibility (ease of development), understandability, maintainability, and robustness. Less emphasis was placed on program speed and size. Fortran 90 has all the features that permit the creation of readable, maintainable, and extensible code. In particular, the ability to create data and program modules with various levels of data hiding allows EnergyPlus to be built out of semi-independent modules. This allows a new EnergyPlus developer to concentrate on programming a single component without having to learn the entire program and data structure.

The EnergyPlus programming style is described in the *EnergyPlus Programming Standard.* The *Programming Standard* should be consulted for details such as variable and subroutine naming conventions. In this document, we will describe the steps a developer must follow to create a new EnergyPlus component model. In particular, we will assume the developer wishes to simulate an HVAC component that cannot yet be modeled by EnergyPlus.

Modules in EnergyPlus

What is a module anyway?

Program Modules

A module is a Fortran 90 programming construct that can be used in various ways. In EnergyPlus, its primary use is to segment a rather large program into smaller, more manageable pieces. Each module is a separate package of source code stored on a separate file. The entire collection of modules, when compiled and linked, forms the executable code of EnergyPlus.

Each module contains source code for closely related data structures and procedures. For instance, the WeatherManager module contains all the weather handling routines in EnergyPlus. The module is contained in the file WeatherManager.f90. Another example is PlantPumps. This module contains all the code to simulate pumps in EnergyPlus. It is contained in file PlantPumps.f90.

Of course dividing a program into modules can be done in various ways. We have attempted to create modules that are as self-contained as possible. The philosophy that has been used in creating EnergyPlus is contained in the Programming Standard reference document. Logically, the modules in EnergyPlus form an inverted tree structure. At the top is EnergyPlus. Just below that are ProcessInput and ManageSimulation. At the bottom are the modules such as HVACDamperComponent that model the actual HVAC components.

Appendix C to this document describes the “manager” structure that has been implemented in EnergyPlus.

Data Only Modules

EnergyPlus also uses modules that contain only data. These modules form one of the primary ways data is structured and shared in EnergyPlus. An example is the DataEnvironment module. Many parts of the program need access to the outdoor conditions. All of that data is encapsulated in DataEnvironment. Modules that need this data obtain access through a Fortran USE statement. Without such access, modules cannot use or change this data.

What is a module developer?

A module developer is someone who is going to add to the simulation capabilities of EnergyPlus. Someone, for instance, who is interested in adding code to model a new type of HVAC equipment. The most straightforward way of doing this is to create a new program module – hence the term “module developer”. Another kind of module developer would be the adaptation of an existing “legacy” code to EnergyPlus.

In EnergyPlus, the first step in creating a new component model is to define the input. So, before we discuss modules in more detail, we must first describe the EnergyPlus input.

Input Concepts

In EnergyPlus, input and output are accomplished by means of ASCII (text) files. On the input side, there are two files:

1) the Input Data Dictionary (IDD) that describes the types (classes) of input objects and the data associated with each object;
2) the Input Data File (IDF) that contains all the data for a particular simulation.

Each EnergyPlus module is responsible for getting its own input. Of course, EnergyPlus provides services to the module that make this quite easy. The first task of a module developer is to design and insert a new entry into the Input Data Dictionary.

Input Data Dictionary

An entry in the IDD consists of comma-separated text terminated by a semicolon. For instance:

```
COIL:Water:SimpleHeating,
  A1 , \field Coil Name
       \type alpha
       \reference HeatingCoilName
  A2 , \field Available Schedule
       \type object-list
       \object-list ScheduleNames
  N1 , \field UA of the Coil
       \units W/K
  N2 , \field Max Water Flow Rate of Coil
       \units kg/s
  A3 , \field Coil_Water_Inlet_Node
  A4 , \field Coil_Water_Outlet_Node
  A5 , \field Coil_Air_Inlet_Node
  A6 ; \field Coil_Air_Outlet_Node
```

This entry defines a simple water-heating coil and specifies all of the input data needed to model it. The following rules apply.

1) The first element *COIL:Water:Simple Heating* is the class name (also called a keyword or key). This class name must be unique in the IDD. The maximum length for the class name is 40 characters. Embedded spaces are allowed and are significant.
2) The initial line of an entry must contain a comma or a semicolon.
3) Commas separate fields. They always act as separators – thus there is no way to include a comma in a class name or as part of a data field.
4) Similarly, semicolons are terminators – a semicolon is always interpreted as the end of an EnergyPlus “sentence”. So, avoid embedded semicolons in class names or data fields.
5) Blank lines are allowed.
6) Each line can be up to 500 characters in length.
7) The comment character is an exclamation or a backslash. Anything on a line after an “!” or a “\” is ignored during EnergyPlus input.

The only significant syntax elements are the commas, the semi colon, the N's (denoting numeric data), and the A's (denoting alphanumeric data) and the exclamation and backslash. Everything else including blanks, end of lines, or even text that is not a comma, semicolon, N, or A is ignored. There are several style conventions in use however.

1) Sequence numbers are appended to the letters A or N denoting each data element. Thus, A2 is the second alphanumeric data item and N3 is the third numeric data item.
2) The class name contains a naming convention: *type:subtype:subsubtype*.
3) Backslashes denote specially formatted comments. These comments provide information about the input, such as a description of the item, units, limits, mins & maxes, etc., in a form that can be processed by an input editor or interface. A complete description of the backslash comment format is given at the start of the IDD file and in the *Guide for Interface Developers*.

Overall, the IDD file has very little structure. Generally, a new entry should be placed next to entries describing similar components. *COIL:Water:Simple Heating*, for instance, is grouped with entries describing other water coils.

Summary

The first task for a module developer is to create a new entry in the Input Data Dictionary. This entry defines the data needed to model the new component.

Input Data File

The Input Data File (IDF) is the file containing the data for an actual simulation. This file is also a text (ASCII) file with a syntax "filling in the blanks" of the definitions in the IDD. A portion of an IDF with input data for the hot water coil defined in the IDD example looks like:

```
!Demand Heating Components
    COIL:Water:SimpleHeating,
         Reheat Coil Zone 1,     !Name of cooling coil
         FanAndCoilAvailSched,  !Cooling Coil Schedule
         400.0,                 !UA of the Coil
         1.3,                   !Max Water Flow Rate of Coil kg/sec
         Zone 1 Reheat Water Inlet Node,  !Water side inlet node
         Zone 1 Reheat Water Outlet Node, !Water side outlet node
         Zone 1 Reheat Air Inlet Node,    !Air side inlet node
         Zone 1 Reheat Air Outlet Node;   !Air side outlet node

    COIL:Water:SimpleHeating,
         Reheat Coil Zone 2,     !Name of cooling coil
         FanAndCoilAvailSched,  !Cooling Coil Schedule
         400.0,                 !UA of the Coil
         1.2,                   !Max Water Flow Rate of Coil kg/sec
         Zone 2 Reheat Water Inlet Node,  !Water side inlet node
         Zone 2 Reheat Water Outlet Node, !Water side outlet node
         Zone 2 Reheat Air Inlet Node,    !Air side inlet node
         Zone 2 Reheat Air Outlet Node;   !Air side outlet node

    COIL:Water:SimpleHeating,
         Reheat Coil Zone 3,     !Name of cooling coil
         FanAndCoilAvailSched,  !Cooling Coil Schedule
```

```
          400.0,                 !UA of the Coil
          1.8,                   !Max Water Flow Rate of Coil kg/sec
          Zone 3 Reheat Water Inlet Node,  !Water side inlet node
          Zone 3 Reheat Water Outlet Node, !Water side outlet node
          Zone 3 Reheat Air Inlet Node,    !Air side inlet node
          Zone 3 Reheat Air Outlet Node;   !Air side outlet node
```

Each coil entry begins with the class name (keyword) specifying the type of coil. Next is the coil name – a user (or interface) created name that is unique within the given class. Generally in EnergyPlus, objects within a class are distinguished by unique names. The object name is usually the first data element following the class name. Any alphanumeric data item in the IDF can be up to 40 characters long. Any characters past 40 are truncated (lost). After the object name comes the real data. If we look at the IDD we see that the first data item after the object name is expected to be an alphanumeric – a schedule name. In the IDF, we see the corresponding field is "FanAndCoilAvailSched", the object name of a schedule elsewhere in the IDF file. In EnergyPlus, all references to other data entries (objects) are via object names. The next two data items are numeric: the coil UA and the maximum water mass flow rate. The final four items are again alphanumeric – the names of the coil inlet and outlet nodes. Nodes are used in EnergyPlus to connect HVAC components together into HVAC systems.

The example illustrates the use of comments to create clear input. The IDF is intended to be human readable, largely for development and debugging purposes. Of course, most users will never see an IDF – they will interact with EnergyPlus through a Graphical User Interface (GUI), which will write the IDF for them. However, a module developer is a special kind of user. The module developer will need to create a portion of an IDF by hand very early in the development process in order to begin testing the module under development. Thus, it is important to understand the IDF syntax and to use comments to create readable test IDF files.

Summary

One of the early tasks of a module developer is to create input (most likely by hand) for the new component and to insert it into an existing IDF file in order to test the new component model. The IDF syntax resembles the syntax for the IDD. The data follows the IDD class description. Comments should be used to make the IDF readable.

Input Considerations

The IDD/IDF concept allows the module developer much flexibility. Along with this flexibility comes a responsibility to the overall development of EnergyPlus. Developers must take care not to obstruct other developers with their additions or changes. Major changes in the IDD require collaboration among the developers (both module and interface).

In many cases, the developer may be creating a new model – a new HVAC component, for instance. Then the most straightforward approach is to create a new object class in the IDD with its own unique, self-contained input. This will seldom impact other developers.

In some cases, the developer may be adding a calculation within an existing module or for an existing class of objects. This calculation may require new or different input fields. Then the developer has a number of choices. This section will present some ideas for adding to the IDD that will minimize impact to other developers.

For example, consider the implementation of Other Side Coefficients (OSC) in the IDD. Other side coefficients are a simplification for the surface heat balance and were used mostly in BLAST 2.0 before we had interzone surfaces. We have carried this forward into EnergyPlus for those users that understand and can use it. We'll use it as an example of approaches to adding data items to the IDD. Moreover, we'll try to give some hints on which approaches might be used for future additions.

So, you're adding something to EnergyPlus and it is part of an existing module or object class.. What do you do with your required inputs to your model? There are at least four options:

1) Embed your values in a current object class definition.

2) Put something in the current definition that will trigger a "GetInput" for your values.

3) Put something in the current definition that will signal a "special" case and embed a name (of your item) in the definition (this adds 1 or 2 properties to the object).

4) Just get your input and have each of those inputs reference a named object.

For example, using the OSC option in surfaces, in beta 2 we had

```
A8 , \field Exterior environment
     \type alpha
     \note <for Interzone Surface:Adjacent surface name>
     \note For non-interzone surfaces enter:
     \note ExteriorEnvironment, Ground, or OtherSideCoeff
     \note OSC won't use CTFs

N24, \field User selected Constant Temperature
N25, \field Coefficient modifying the user selected constant
            temperature
N26, \field Coefficient modifying the external dry bulb temperature
N27, \field Coefficient modifying the ground temperature
N28, \field Combined convective/radiative film coefficient
     \note if=0, use other coefficients
N29, \field Coefficient modifying the wind speed term (s/m)
N30, \field Coefficient modifying the zone air temperature part of
            the equation
```

1) We have done option 1: embed the values in the input. (We have also embedded these values in each and every surface derived type (internal data structure) but that can be discussed elsewhere).

When to use: It makes sense to embed these values when each and every object (SURFACE) needs these values (e.g. we need to specify Vertices for Every Surface -- so these clearly should be embedded).

After beta 2, the definition of Surfaces was changed Obviously option 1 was not a good choice for the OCF data: the data would be rarely used. Our other options were:

2) Obviously the ExteriorEnvironment field will remain (but its name was to Outside Face Environment).

However, we do not want to embed the values for OtherSideCoef in the Surface items. So, if the ExteriorEnvironment continues to reference OtherSideCoef, we can easily trigger a "GetInput" for them. An additional object class would be necessary for this case.

```
OtherSideCoef,  A1, \field name of OtherSideCoef,
  A2, \field SurfaceName (reference to surface using OSC)
  ....
```

When to use: This option can be used for many cases. The same object definition will work for option 4 below. Obviously, if there is not a convenient trigger in SURFACE but you want to add a feature, this would let you do it without embedding it in the Surface Definition. If there is a trigger, such as exists with the ExteriorEnvironment, the A2 field might not be needed. This approach would become a bit cumbersome if you expected there to be a lot of these or if there were a one to many relationship (i.e. a single set of OSCs could be used for many surfaces). Nevertheless, the approach provides a convenient "data check"/cross reference that can be validated inside the code.

3) We could also have the SURFACE definition reference an OSC name (in this instance).

So, we'd add a field to the Surface that would be the name in the OtherSideCoef object above. Then, the OtherSideCoef objects wouldn't need a Surface Name. This is the most straightforward approach: including data in one object by referencing another, and it was the approach chosen for the redefined Surface class.

When to use: when there is a set of parameters that would be used extensively, then this would provide a name for those. If hand editing, then you only would need to change one set of these parameters rather than having to go through many. Of course, the OtherSideCoef object wouldn't also have to have the true numbers but could reference yet a third named object...... (starting to get messy).

4) We could have the OtherSideCoef object as above and just "get" it as a matter of course. (In the case where we don't have a convenient trigger such as ExteriorEnvironment).

When to use: Note that the same structure for 2 works here too. It's just not triggered (to get the input) by a value in the other object (SURFACE).

Summary

There are several approaches to adding items to the IDD. Developers need to consider impacts to other developers and users early in the implementation planning.

Module Structure

Let us assume that the novice EnergyPlus developer wishes to model a new HVAC component called *NewHVACComponent*. Right at the start there is a choice to make: whether to insert the new model into an existing module or to create an entirely new EnergyPlus component simulation model. Creating a new module is the easier option to explain, implement and test. We will discuss this option in this document. The discussion should also impart enough information to allow a new developer to insert a model into an existing EnergyPlus module if that option is chosen.

Module Outline

The developer will create a new file NewHVACComponent.f90. The file will contain the following elements:

MODULE NewHVACComponent

Documentation: Fortran comments describing and documenting the module. Included are sections showing module author, module creation date, date modified and modification author;

USE Statements*:* Fortran statements naming other modules that this module can access, either for data or for routines.

Module Datastructure Definitions: Using the Fortran TYPE statement define the data structures needed in the module that will not be available from other modules. Define all module level variables that will be needed.

CONTAINS

SUBROUTINE *SimNewHVACComponent*

This routine selects the individual component being simulated and calls the other module subroutines that do the real work. This routine is the only routine in the module that is accessible outside the module (*PUBLIC*). All other routines in the module are *PRIVATE* and are only callable within the module. This routine is sometimes called the “driver” routine for the module.

END SUBROUTINE *SimNewHVACComponent*

SUBROUTINE *GetNewHVACComponentInput*

This routine uses the “get” routines from the InputProcessor module to obtain input for NewHVACComponent. The module data arrays are allocated and the data is moved into the arrays.

END SUBROUTINE *GetNewHVACComponentInput*

SUBROUTINE *InitNewHVACComponent*

This routine performs whatever initialization calculations that may be needed at various points in the simulation. For instance, some calculations may only need to be done once; some may need to be done at the start of each simulation weather period; some at the start of each HVAC simulation time step; and some at the start of each loop solution. This routine also transfers data from the component inlet nodes to the component data arrays every time the component is simulated, in preparation for the actual component simulation.

END SUBROUTINE *InitNewHVACComponent*

SUBROUTINE *CalcNewHVACComponent*

This routine does the actual calculations to simulate the performance of the component. Only calculation is done – there is no moving of data from or to input or output areas. There may be more than one "*CALC*" subroutine if more than one component is being modeled within this module.

END SUBROUTINE *CalcNewHVACComponent*

SUBROUTINE *UpdateNewHVACComponent*

This routine moves the results of the "*Calc*" routine(s) to the component outlet nodes.

END SUBROUTINE *UpdateNewHVACComponent*

SUBROUTINE *ReportNewHVACComponent*

This routine performs any special calculations that are needed purely for reporting purposes.

END SUBROUTINE *ReportNewHVACComponent*

END MODULE *NewHVACComponent*

Module Example

```
Module Fans
  ! Module containing the fan simulation routines

  ! MODULE INFORMATION:
  !       AUTHOR         Richard J. Liesen
  !       DATE WRITTEN   April 1998
  !       MODIFIED       na
  !       RE-ENGINEERED  na

  ! PURPOSE OF THIS MODULE:
  ! To encapsulate the data and algorithms required to
  ! manage the Fan System Component

  ! REFERENCES: none

  ! OTHER NOTES: none

  ! USE STATEMENTS:
  ! Use statements for data only modules
USE DataLoopNode
USE DataGlobals,     ONLY: SetupOutputVariable, BeginSimFlag, BeginEnvrnFlag, BeginDayFlag, &
                           MaxNameLength, &
                           ShowWarningError, ShowFatalError, ShowSevereError
Use DataEnvironment, ONLY: OutBaroPress

  ! Use statements for access to subroutines in other modules
USE ScheduleManager

IMPLICIT NONE         ! Enforce explicit typing of all variables

PRIVATE ! Everything private unless explicitly made public

  !MODULE PARAMETER DEFINITIONS
  ! na

  ! DERIVED TYPE DEFINITIONS
TYPE FanEquipConditions
  CHARACTER(len=MaxNameLength) :: FanName  ! Name of the fan
  CHARACTER(len=MaxNameLength) :: FanType  ! Type of Fan ie. Simple, Vane axial, Centrifugal, etc.
  CHARACTER(len=MaxNameLength) :: Schedule ! Fan Operation Schedule
  CHARACTER(len=MaxNameLength) :: Control  ! ie. Const Vol, Variable Vol
  Integer      :: SchedPtr ! Pointer to the correct schedule
  REAL         :: InletAirMassFlowRate  !MassFlow through the Fan being Simulated [kg/Sec]
  REAL         :: OutletAirMassFlowRate
  Real         :: MaxAirFlowRate  !Max Specified Volume Flow Rate of Fan [m^3/sec]
  Real         :: MinAirFlowRate  !Min Specified Volume Flow Rate of Fan [m^3/sec]
  REAL         :: MaxAirMassFlowRate ! Max flow rate of fan in kg/sec
  REAL         :: MinAirMassFlowRate ! Min flow rate of fan in kg/sec
  REAL         :: InletAirTemp
  REAL         :: OutletAirTemp
  REAL         :: InletAirHumRat
  REAL         :: OutletAirHumRat
  REAL         :: InletAirEnthalpy
  REAL         :: OutletAirEnthalpy
  REAL         :: FanPower             !Power of the Fan being Simulated [kW]
  REAL         :: FanEnergy            !Fan energy in [kJ]
  REAL         :: DeltaTemp            !Temp Rise across the Fan [C]
  REAl         :: DeltaPress           !Delta Pressure Across the Fan [N/M^2]
  REAL         :: FanEff               !Fan total efficiency; motor and mechanical
  REAL         :: MotEff               !Fan motor efficiency
  REAL         :: MotInAirFrac         !Fraction of motor heat entering air stream
  REAL, Dimension(5):: FanCoeff        !Fan Part Load Coefficients to match fan type
  ! Mass Flow Rate Control Variables
  REAL         :: MassFlowRateMaxAvail
  REAL         :: MassFlowRateMinAvail
  INTEGER      :: InletNodeNum
  INTEGER      :: OutletNodeNum
END TYPE FanEquipConditions
```

```
  !MODULE VARIABLE DECLARATIONS:
  INTEGER :: NumFans     ! The Number of Fans found in the Input
  TYPE (FanEquipConditions), ALLOCATABLE, DIMENSION(:) :: Fan

! Subroutine Specifications for the Module
          ! Driver/Manager Routines
Public  SimulateFanComponents

          ! Get Input routines for module
PRIVATE GetFanInput

          ! Initialization routines for module
PRIVATE InitFan

          ! Algorithms for the module
Private SimSimpleFan
PRIVATE SimVariableVolumeFan
PRIVATE SimZoneExhaustFan


          ! Update routine to check convergence and update nodes
Private UpdateFan

          ! Reporting routines for module
Private ReportFan


CONTAINS

! MODULE SUBROUTINES:
!*************************************************************************
SUBROUTINE SimulateFanComponents(CompName,FirstHVACIteration)

          ! SUBROUTINE INFORMATION:
          !       AUTHOR         Richard Liesen
          !       DATE WRITTEN   February 1998
          !       MODIFIED       na
          !       RE-ENGINEERED  na

          ! PURPOSE OF THIS SUBROUTINE:
          ! This subroutine manages Fan component simulation.

          ! METHODOLOGY EMPLOYED:
          ! na

          ! REFERENCES:
          ! na

          ! USE STATEMENTS:
  USE InputProcessor, ONLY: FindItemInList

  IMPLICIT NONE    ! Enforce explicit typing of all variables in this routine

          ! SUBROUTINE ARGUMENT DEFINITIONS:
  LOGICAL,      INTENT (IN):: FirstHVACIteration
  CHARACTER(len=*), INTENT(IN) :: CompName

          ! SUBROUTINE PARAMETER DEFINITIONS:
          ! na

          ! INTERFACE BLOCK SPECIFICATIONS

          ! DERIVED TYPE DEFINITIONS
          ! na

          ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
  INTEGER             :: FanNum     ! current fan number
  Logical :: GetInputFlag = .True.  ! Flag set to make sure you get input once

          ! FLOW:
```

```
  ! Obtains and Allocates fan related parameters from input file
  IF (GetInputFlag) THEN  !First time subroutine has been entered
    CALL GetFanInput
    GetInputFlag=.false.
  End If


  ! Find the correct FanNumber with the AirLoop & CompNum from AirLoop Derived Type
  !FanNum = AirLoopEquip(AirLoopNum)%ComponentOfTypeNum(CompNum)
  ! Find the correct WaterCoilNumber with the Coil Name
  FanNum = FindItemInList(CompName,Fan%FanName,NumFans)
  IF (FanNum == 0) THEN
    CALL ShowFatalError('Fan not found='//TRIM(CompName))
  ENDIF

  ! With the correct FanNum Initialize
  CALL InitFan(FanNum,FirstHVACIteration)  ! Initialize all fan related parameters

  ! Calculate the Correct Fan Model with the current FanNum
  If(Fan(FanNum)%FanType == 'SIMPLE' .and. &
     Fan(FanNum)%Control == 'CONSTVOLUME') Then
    Call SimSimpleFan(FanNum)
  Else IF(Fan(FanNum)%FanType == 'SIMPLE' .and. &
     Fan(FanNum)%Control == 'VARIABLEVOLUME' .OR. &
     Fan(FanNum)%Control == 'ONOFF') Then
    Call SimVariableVolumeFan(FanNum)
  Else If(Fan(FanNum)%FanType == 'ZONE EXHAUST FAN') Then
    Call SimZoneExhaustFan(FanNum)
  End If

  ! Update the current fan to the outlet nodes
  Call UpdateFan(FanNum)

  ! Report the current fan
  Call ReportFan(FanNum)

  RETURN

END SUBROUTINE SimulateFanComponents


! Get Input Section of the Module
!******************************************************************************
SUBROUTINE GetFanInput

          ! SUBROUTINE INFORMATION:
          !       AUTHOR         Richard Liesen
          !       DATE WRITTEN   April 1998
          !       MODIFIED       na
          !       RE-ENGINEERED  na

          ! PURPOSE OF THIS SUBROUTINE:
          ! Obtains input data for fans and stores it in fan data structures

          ! METHODOLOGY EMPLOYED:
          ! Uses "Get" routines to read in data.

          ! REFERENCES:
          ! na

          ! USE STATEMENTS:
    USE InputProcessor
    USE NodeInputManager

  IMPLICIT NONE    ! Enforce explicit typing of all variables in this routine

          ! SUBROUTINE ARGUMENT DEFINITIONS:
          ! na

          ! SUBROUTINE PARAMETER DEFINITIONS:
```

```
          ! na

          ! INTERFACE BLOCK SPECIFICATIONS
          ! na

          ! DERIVED TYPE DEFINITIONS
          ! na

          ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
!    INTEGER :: NumAirLoops ! Read Input for the Total Number of Air Loops
!    INTEGER :: AirLoopNum  ! primary air loop index
!    INTEGER :: CompNum
    INTEGER :: FanNum      ! The fan that you are currently loading input into
    INTEGER :: NumSimpFan  ! The number of Simple Const Vol Fans
    INTEGER :: NumVarVolFan ! The number of Simple Variable Vol Fans
    INTEGER :: NumOnOff     ! The number of Simple on-off Fans
    INTEGER :: NumZoneExhFan
    INTEGER :: SimpFanNum
    INTEGER :: OnOffFanNum
    INTEGER :: VarVolFanNum
    INTEGER :: ExhFanNum
    INTEGER :: NumAlphas
    INTEGER :: NumNums
    INTEGER :: IOSTAT
    REAL, DIMENSION(11) :: NumArray
    CHARACTER(len=MaxNameLength), DIMENSION(4) :: AlphArray
    INTEGER :: NumNodes
    INTEGER, DIMENSION(25) :: NodeNums
    LOGICAL :: ErrorsFound = .false.   ! If errors detected in input
    LOGICAL       :: IsNotOK               ! Flag to verify name
    LOGICAL       :: IsBlank               ! Flag for blank name

          ! Flow
    NumSimpFan   = GetNumObjectsFound('FAN:SIMPLE:CONSTVOLUME')
    NumVarVolFan = GetNumObjectsFound('FAN:SIMPLE:VARIABLEVOLUME')
    NumOnOff = GetNumObjectsFound('FAN:SIMPLE:ONOFF')
    NumZoneExhFan = GetNumObjectsFound('ZONE EXHAUST FAN')
    NumFans = NumSimpFan + NumVarVolFan + NumZoneExhFan+NumOnOff
    IF (NumFans.GT.0) ALLOCATE(Fan(NumFans))
    ! Initialize fan data structures
    Fan%FanName = ' '
    Fan%FanType = ' '
    Fan%Schedule = ' '
    Fan%Control = ' '
    Fan%SchedPtr = 0
    Fan%InletAirMassFlowRate = 0.0
    Fan%OutletAirMassFlowRate = 0.0
    Fan%MaxAirFlowRate = 0.0
    Fan%MinAirFlowRate = 0.0
    Fan%MaxAirMassFlowRate = 0.0
    Fan%MinAirMassFlowRate = 0.0
    Fan%InletAirTemp = 0.0
    Fan%OutletAirTemp = 0.0
    Fan%InletAirHumRat = 0.0
    Fan%OutletAirHumRat = 0.0
    Fan%InletAirEnthalpy = 0.0
    Fan%OutletAirEnthalpy = 0.0
    Fan%FanPower = 0.0
    Fan%FanEnergy = 0.0
    Fan%DeltaTemp = 0.0
    Fan%DeltaPress = 0.0
    Fan%FanEff = 0.0
    Fan%MotEff = 0.0
    Fan%MotInAirFrac = 0.0
    Fan%FanCoeff(1) = 0.0
    Fan%FanCoeff(2) = 0.0
    Fan%FanCoeff(3) = 0.0
    Fan%FanCoeff(4) = 0.0
    Fan%FanCoeff(5) = 0.0
    Fan%MassFlowRateMaxAvail = 0.0
    Fan%MassFlowRateMinAvail = 0.0
    Fan%InletNodeNum = 0
    Fan%OutletNodeNum = 0

      DO SimpFanNum = 1,  NumSimpFan
        FanNum = SimpFanNum
        CALL GetObjectItem('FAN:SIMPLE:CONSTVOLUME',SimpFanNum,AlphArray, &
                           NumAlphas,NumArray,NumNums,IOSTAT)
        IsNotOK=.false.
        IsBlank=.false.
        CALL VerifyName(AlphArray(1),Fan%FanName,FanNum-1,IsNotOK,IsBlank, &
                        'FAN:SIMPLE:CONSTVOLUME Name')
        IF (IsNotOK) THEN
          ErrorsFound=.true.
          IF (IsBlank) AlphArray(1)='xxxxx'
        ENDIF
        Fan(FanNum)%FanName = AlphArray(1)
        Fan(FanNum)%FanType = 'SIMPLE'
        Fan(FanNum)%Schedule = AlphArray(2)
        Fan(FanNum)%SchedPtr =GetScheduleIndex(AlphArray(2))
        IF (Fan(FanNum)%SchedPtr == 0) THEN
          CALL ShowSevereError('FAN:SIMPLE:CONSTVOLUME, Schedule not found='//TRIM(AlphArray(2)))
          ErrorsFound=.true.
        ENDIF
        Fan(FanNum)%Control = 'CONSTVOLUME'

        Fan(FanNum)%FanEff        = NumArray(1)
        Fan(FanNum)%DeltaPress    = NumArray(2)
        Fan(FanNum)%MaxAirFlowRate= NumArray(3)
        Fan(FanNum)%MotEff     = NumArray(4)
        Fan(FanNum)%MotInAirFrac  = NumArray(5)
        Fan(FanNum)%MinAirFlowRate= 0.0

        CALL GetNodeNums(AlphArray(3),NumNodes,NodeNums)
        IF (NumNodes > 1) THEN
          CALL ShowSevereError('Fan:Simple:ConstVolume:Only 1st Node used from:' &
                                //TRIM(AlphArray(3)))
          ErrorsFound=.true.
        ENDIF
        Fan(FanNum)%InletNodeNum  = NodeNums(1)
        CALL GetNodeNums(AlphArray(4),NumNodes,NodeNums)
        IF (NumNodes > 1) THEN
          CALL ShowSevereError('Fan:Simple:ConstVolume:Only 1st Node used from:' &
                               //TRIM(AlphArray(4)))
          ErrorsFound=.true.
        ENDIF
        Fan(FanNum)%OutletNodeNum = NodeNums(1)

      END DO   ! end Number of Simple FAN Loop


      DO VarVolFanNum = 1,  NumVarVolFan
        FanNum = NumSimpFan + VarVolFanNum
        CALL GetObjectItem('FAN:SIMPLE:VARIABLEVOLUME',VarVolFanNum,AlphArray, &
                           NumAlphas,NumArray,NumNums,IOSTAT)
        IsNotOK=.false.
        IsBlank=.false.
        CALL VerifyName(AlphArray(1),Fan%FanName,FanNum-1,IsNotOK,IsBlank, &
                        'FAN:SIMPLE:VARIABLEVOLUME Name')
        IF (IsNotOK) THEN
          ErrorsFound=.true.
          IF (IsBlank) AlphArray(1)='xxxxx'
        ENDIF
        Fan(FanNum)%FanName = AlphArray(1)
        Fan(FanNum)%FanType = 'SIMPLE'
        Fan(FanNum)%Schedule = AlphArray(2)
        Fan(FanNum)%SchedPtr =GetScheduleIndex(AlphArray(2))
        IF (Fan(FanNum)%SchedPtr == 0) THEN
          CALL ShowSevereError('FAN:SIMPLE:VARIABLEVOLUME, Schedule not found=' &
                               //TRIM(AlphArray(2)))
          ErrorsFound=.true.
        ENDIF
        Fan(FanNum)%Control = 'VARIABLEVOLUME'

        Fan(FanNum)%FanEff        = NumArray(1)
        Fan(FanNum)%DeltaPress    = NumArray(2)
        Fan(FanNum)%MaxAirFlowRate= NumArray(3)
        Fan(FanNum)%MinAirFlowRate= NumArray(4)
        Fan(FanNum)%MotEff        = NumArray(5)
        Fan(FanNum)%MotInAirFrac  = NumArray(6)
        Fan(FanNum)%FanCoeff(1)   = NumArray(7)
        Fan(FanNum)%FanCoeff(2)   = NumArray(8)
        Fan(FanNum)%FanCoeff(3)   = NumArray(9)
        Fan(FanNum)%FanCoeff(4)   = NumArray(10)
        Fan(FanNum)%FanCoeff(5)   = NumArray(11)
        CALL GetNodeNums(AlphArray(3),NumNodes,NodeNums)
        IF (NumNodes > 1) THEN
          CALL ShowSevereError('Fan:Simple:VariableVolume:Only 1st Node used from:' &
                               //TRIM(AlphArray(3)))
          ErrorsFound=.true.
        ENDIF
        Fan(FanNum)%InletNodeNum  = NodeNums(1)
        CALL GetNodeNums(AlphArray(4),NumNodes,NodeNums)
        IF (NumNodes > 1) THEN
          CALL ShowSevereError('Fan:Simple:VariableVolume:Only 1st Node used from:' &
                               //TRIM(AlphArray(4)))
          ErrorsFound=.true.
        ENDIF
        Fan(FanNum)%OutletNodeNum = NodeNums(1)

      END DO   ! end Number of Variable Volume FAN Loop

      DO ExhFanNum = 1,  NumZoneExhFan
        FanNum = NumSimpFan + NumVarVolFan + ExhFanNum
        CALL GetObjectItem('ZONE EXHAUST FAN',ExhFanNum,AlphArray, &
                           NumAlphas,NumArray,NumNums,IOSTAT)
        IsNotOK=.false.
        IsBlank=.false.
        CALL VerifyName(AlphArray(1),Fan%FanName,FanNum-1,IsNotOK,IsBlank,'ZONE EXHAUST FAN Name')
        IF (IsNotOK) THEN
          ErrorsFound=.true.
          IF (IsBlank) AlphArray(1)='xxxxx'
        ENDIF
        Fan(FanNum)%FanName = AlphArray(1)
        Fan(FanNum)%FanType = 'ZONE EXHAUST FAN'
        Fan(FanNum)%Schedule = AlphArray(2)
        Fan(FanNum)%SchedPtr =GetScheduleIndex(AlphArray(2))
        IF (Fan(FanNum)%SchedPtr == 0) THEN
          CALL ShowSevereError('ZONE EXHAUST FAN, Schedule not found='//TRIM(AlphArray(2)))
          ErrorsFound=.true.
        ENDIF
        Fan(FanNum)%Control = 'CONSTVOLUME'

        Fan(FanNum)%FanEff        = NumArray(1)
        Fan(FanNum)%DeltaPress    = NumArray(2)
        Fan(FanNum)%MaxAirFlowRate= NumArray(3)
        Fan(FanNum)%MotEff     = 1.0
        Fan(FanNum)%MotInAirFrac  = 1.0
        Fan(FanNum)%MinAirFlowRate= 0.0

        CALL GetNodeNums(AlphArray(3),NumNodes,NodeNums)
        IF (NumNodes > 1) THEN
          CALL ShowSevereError('Fan:Simple:ConstVolume:Only 1st Node used from:' &
                               //TRIM(AlphArray(3)))
          ErrorsFound=.true.
        ENDIF
        Fan(FanNum)%InletNodeNum  = NodeNums(1)
        CALL GetNodeNums(AlphArray(4),NumNodes,NodeNums)
        IF (NumNodes > 1) THEN
          CALL ShowSevereError('Fan:Simple:ConstVolume:Only 1st Node used from:' &
                               //TRIM(AlphArray(4)))
          ErrorsFound=.true.
        ENDIF
```

```
      Fan(FanNum)%OutletNodeNum = NodeNums(1)

    END DO   ! end of Zone Exhaust Fan loop

    DO OnOffFanNum = 1,  NumOnOff
      FanNum = NumSimpFan + NumVarVolFan + NumZoneExhFan + OnOffFanNum
      CALL GetObjectItem('FAN:SIMPLE:ONOFF',OnOffFanNum,AlphArray, &
                         NumAlphas,NumArray,NumNums,IOSTAT)
      IsNotOK=.false.
      IsBlank=.false.
      CALL VerifyName(AlphArray(1),Fan%FanName,FanNum-1,IsNotOK,IsBlank,'FAN:SIMPLE:ONOFF Name')
      IF (IsNotOK) THEN
        ErrorsFound=.true.
        IF (IsBlank) AlphArray(1)='xxxxx'
      ENDIF
      Fan(FanNum)%FanName = AlphArray(1)
      Fan(FanNum)%FanType = 'SIMPLE'
      Fan(FanNum)%Schedule = AlphArray(2)
      Fan(FanNum)%SchedPtr =GetScheduleIndex(AlphArray(2))
      IF (Fan(FanNum)%SchedPtr == 0) THEN
        CALL ShowSevereError('FAN:SIMPLE:ONOFF, Schedule not found='//TRIM(AlphArray(2)))
        ErrorsFound=.true.
      ENDIF
      Fan(FanNum)%Control = 'ONOFF'

      Fan(FanNum)%FanEff        = NumArray(1)
      Fan(FanNum)%DeltaPress    = NumArray(2)
      Fan(FanNum)%MaxAirFlowRate= NumArray(3)
      Fan(FanNum)%MotEff     = NumArray(4)
      Fan(FanNum)%MotInAirFrac  = NumArray(5)
      Fan(FanNum)%MinAirFlowRate= 0.0
      Fan%FanCoeff(1) = 0.0
      Fan%FanCoeff(2) = 1.0
      Fan%FanCoeff(3) = 0.0
      Fan%FanCoeff(4) = 0.0
      Fan%FanCoeff(5) = 0.0


      CALL GetNodeNums(AlphArray(3),NumNodes,NodeNums)
      IF (NumNodes > 1) THEN
        CALL ShowSevereError('Fan:Simple:OnOff:Only 1st Node used from:'//TRIM(AlphArray(3)))
        ErrorsFound=.true.
      ENDIF
      Fan(FanNum)%InletNodeNum  = NodeNums(1)
      CALL GetNodeNums(AlphArray(4),NumNodes,NodeNums)
      IF (NumNodes > 1) THEN
        CALL ShowSevereError('Fan:Simple:OnOff:Only 1st Node used from:'//TRIM(AlphArray(4)))
        ErrorsFound=.true.
      ENDIF
      Fan(FanNum)%OutletNodeNum = NodeNums(1)

    END DO   ! end Number of Simple  ON-OFF FAN Loop


    IF (ErrorsFound) THEN
      CALL ShowFatalError('Errors found in getting Fan input')
    ENDIF

    Do FanNum=1,NumFans
           ! Setup Report variables for the Fans
     CALL SetupOutputVariable('Fan Electric Power[W]', Fan(FanNum)%FanPower, &
                              'System','Average',Fan(FanNum)%FanName)
     CALL SetupOutputVariable('Fan Delta Temp[C]', Fan(FanNum)%DeltaTemp, &
                              'System','Average',Fan(FanNum)%FanName)
     CALL SetupOutputVariable('Fan Electric Consumption[J]', Fan(FanNum)%FanEnergy, &
                              'System','Sum',Fan(FanNum)%FanName, &
                               ResourceTypeKey='Electric',EndUseKey='Fans',GroupKey='System')

    END DO

  RETURN
```

```
END SUBROUTINE GetFanInput

! End of Get Input subroutines for the HB Module
!******************************************************************************



! Beginning Initialization Section of the Module
!******************************************************************************

SUBROUTINE InitFan(FanNum,FirstHVACIteration)

          ! SUBROUTINE INFORMATION:
          !       AUTHOR         Richard J. Liesen
          !       DATE WRITTEN   February 1998
          !       MODIFIED       na
          !       RE-ENGINEERED  na

          ! PURPOSE OF THIS SUBROUTINE:
          ! This subroutine is for initializations of the Fan Components.

          ! METHODOLOGY EMPLOYED:
          ! Uses the status flags to trigger initializations.

          ! REFERENCES:
          ! na

          ! USE STATEMENTS:
          ! na

  IMPLICIT NONE    ! Enforce explicit typing of all variables in this routine

          ! SUBROUTINE ARGUMENT DEFINITIONS:
  LOGICAL, INTENT (IN):: FirstHVACIteration
  Integer, Intent(IN) :: FanNum

          ! SUBROUTINE PARAMETER DEFINITIONS:
          ! na

          ! INTERFACE BLOCK SPECIFICATIONS
          ! na

          ! DERIVED TYPE DEFINITIONS
          ! na

          ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
  Integer             :: InletNode
  Integer             :: OutletNode
  Integer             :: FanIndex
  Integer             :: InNode
  Integer             :: OutNode
  Real                :: RhoAir
  LOGICAL             :: MyEnvrnFlag=.true.
  Real RhoAirFN

          ! FLOW:

! Do the Begin Simulation initializations
  ! none

! Do the Begin Environment initializations
  IF (BeginEnvrnFlag .and. MyEnvrnFlag) THEN

    Do FanIndex = 1, NumFans
       !For all Fan inlet nodes convert the Volume flow to a mass flow
       InNode = Fan(FanIndex)%InletNodeNum
       OutNode = Fan(FanIndex)%OutletNodeNum
       RhoAir = RhoAirFn(OutBaroPress,25.,0.014)

       !Change the Volume Flow Rates to Mass Flow Rates
```

```
      Fan(FanIndex)%MaxAirMassFlowRate = Fan(FanIndex)%MaxAirFlowRate *  RhoAir
      Fan(FanIndex)%MinAirMassFlowRate = Fan(FanIndex)%MinAirFlowRate *  RhoAir


      !Init the Node Control variables
      Node(OutNode)%MassFlowRateMax      = Fan(Fanindex)%MaxAirMassFlowRate
      Node(OutNode)%MassFlowRateMin      = Fan(Fanindex)%MinAirMassFlowRate


      !Initialize all report variables to a known state at beginning of simulation
      Fan(Fanindex)%FanPower = 0.0
      Fan(Fanindex)%DeltaTemp = 0.0
      Fan(Fanindex)%FanEnergy = 0.0

    End Do
    MyEnvrnFlag = .FALSE.
  END IF

  IF (.not. BeginEnvrnFlag) THEN
    MyEnvrnFlag=.true.
  ENDIF

  ! Do the Begin Day initializations
    ! none

  ! Do the begin HVAC time step initializations
    ! none

  ! Do the following initializations (every time step): This should be the info from
  ! the previous components outlets or the node data in this section.

  ! Do a check and make sure that the max and min available(control) flow is
  ! between the physical max and min for the Fan while operating.

  InletNode = Fan(FanNum)%InletNodeNum
  OutletNode = Fan(FanNum)%OutletNodeNum

  Fan(FanNum)%MassFlowRateMaxAvail = MIN(Node(OutletNode)%MassFlowRateMax, &
                                         Node(InletNode)%MassFlowRateMaxAvail)
  Fan(FanNum)%MassFlowRateMinAvail = MIN(MAX(Node(OutletNode)%MassFlowRateMin, &
                                         Node(InletNode)%MassFlowRateMinAvail), &
                                         Node(InletNode)%MassFlowRateMaxAvail)

  ! Load the node data in this section for the component simulation
  !
  !First need to make sure that the massflowrate is between the max and min avail.
  IF (Fan(FanNum)%FanType .NE. 'ZONE EXHAUST FAN') THEN
    Fan(FanNum)%InletAirMassFlowRate = Min(Node(InletNode)%MassFlowRate, &
                                        Fan(FanNum)%MassFlowRateMaxAvail)
    Fan(FanNum)%InletAirMassFlowRate = Max(Fan(FanNum)%InletAirMassFlowRate, &
                                        Fan(FanNum)%MassFlowRateMinAvail)
  ELSE  ! zone exhaust fans - always run at the max
    Fan(FanNum)%MassFlowRateMaxAvail = Fan(FanNum)%MaxAirMassFlowRate
    Fan(FanNum)%MassFlowRateMinAvail = 0.0
    Fan(FanNum)%InletAirMassFlowRate = Fan(FanNum)%MassFlowRateMaxAvail
  END IF

  !Then set the other conditions
  Fan(FanNum)%InletAirTemp         = Node(InletNode)%Temp
  Fan(FanNum)%InletAirHumRat       = Node(InletNode)%HumRat
  Fan(FanNum)%InletAirEnthalpy     = Node(InletNode)%Enthalpy

  RETURN

END SUBROUTINE InitFan

! End Initialization Section of the Module
!******************************************************************************
```

```
! Begin Algorithm Section of the Module
!******************************************************************************
SUBROUTINE SimSimpleFan(FanNum)

          ! SUBROUTINE INFORMATION:
          !       AUTHOR         Unknown
          !       DATE WRITTEN   Unknown
          !       MODIFIED       na
          !       RE-ENGINEERED  na

          ! PURPOSE OF THIS SUBROUTINE:
          ! This subroutine simulates the simple constant volume fan.

          ! METHODOLOGY EMPLOYED:
          ! Converts design pressure rise and efficiency into fan power and temperature rise
          ! Constant fan pressure rise is assumed.

          ! REFERENCES:
          ! ASHRAE HVAC 2 Toolkit, page 2-3 (FANSIM)

          ! USE STATEMENTS:
          ! na

  IMPLICIT NONE    ! Enforce explicit typing of all variables in this routine

          ! SUBROUTINE ARGUMENT DEFINITIONS:
   Integer, Intent(IN) :: FanNum

          ! SUBROUTINE PARAMETER DEFINITIONS:
          ! na

          ! INTERFACE BLOCK SPECIFICATIONS
          ! na

          ! DERIVED TYPE DEFINITIONS
          ! na

          ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
      Real RhoAir
      Real DeltaPress  ! [N/M^2]
      Real FanEff
      Real MassFlow    ! [kg/sec]
      Real Tin         ! [C]
      Real Win
      Real FanShaftPower ! power delivered to fan shaft
      Real PowerLossToAir ! fan and motor loss to air stream (watts)
      Real, External :: Psythw   ! calculate temperature from enthalpy and humidity ratio
      Real, External :: rhoairfn

   DeltaPress = Fan(FanNum)%DeltaPress
   FanEff     = Fan(FanNum)%FanEff

   ! For a Constant Volume Simple Fan the Max Flow Rate is the Flow Rate for the fan
   Tin        = Fan(FanNum)%InletAirTemp
   Win        = Fan(FanNum)%InletAirHumRat
   RhoAir     = RhoAirFn(OutBaroPress,Tin,Win)
   MassFlow   = MIN(Fan(FanNum)%InletAirMassFlowRate,Fan(FanNum)%MaxAirMassFlowRate)
   MassFlow   = MAX(MassFlow,Fan(FanNum)%MinAirMassFlowRate)
   !
   !Determine the Fan Schedule for the Time step
 If((GetCurrentScheduleValue(Fan(FanNum)%SchedPtr) .gt. 0.0) .and. &
        (Massflow .gt. 0.0)) Then
   !Fan is operating
   Fan(FanNum)%FanPower = MassFlow*DeltaPress/(FanEff*RhoAir) ! total fan power
   FanShaftPower = Fan(FanNum)%MotEff * Fan(FanNum)%FanPower  ! power delivered to shaft
   PowerLossToAir = FanShaftPower + (Fan(FanNum)%FanPower - FanShaftPower) * &
     Fan(FanNum)%MotInAirFrac
   Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy + PowerLossToAir/MassFlow
   ! This fan does not change the moisture or Mass Flow across the component
   Fan(FanNum)%OutletAirHumRat       = Fan(FanNum)%InletAirHumRat
   Fan(FanNum)%OutletAirMassFlowRate = MassFlow
   Fan(FanNum)%OutletAirTemp = Psythw(Fan(FanNum)%OutletAirEnthalpy,Fan(FanNum)%OutletAirHumRat)

 Else
   !Fan is off and not operating no power consumed and mass flow rate.
   Fan(FanNum)%FanPower = 0.0
   FanShaftPower = 0.0
   PowerLossToAir = 0.0
   Fan(FanNum)%OutletAirMassFlowRate = 0.0
   Fan(FanNum)%OutletAirHumRat       = Fan(FanNum)%InletAirHumRat
   Fan(FanNum)%OutletAirEnthalpy     = Fan(FanNum)%InletAirEnthalpy
   Fan(FanNum)%OutletAirTemp = Fan(FanNum)%InletAirTemp
   ! Set the Control Flow variables to 0.0 flow when OFF.
   Fan(FanNum)%MassFlowRateMaxAvail = 0.0
   Fan(FanNum)%MassFlowRateMinAvail = 0.0

 End If

 RETURN
END SUBROUTINE SimSimpleFan


SUBROUTINE SimVariableVolumeFan(FanNum)

          ! SUBROUTINE INFORMATION:
          !       AUTHOR         Unknown
          !       DATE WRITTEN   Unknown
          !       MODIFIED       na
          !       RE-ENGINEERED  na

          ! PURPOSE OF THIS SUBROUTINE:
          ! This subroutine simulates the simple variable volume fan.

          ! METHODOLOGY EMPLOYED:
          ! Converts design pressure rise and efficiency into fan power and temperature rise
          ! Constant fan pressure rise is assumed.
          ! Uses curves of fan power fraction vs. fan part load to determine fan power at
          ! off design conditions.

          ! REFERENCES:
          ! ASHRAE HVAC 2 Toolkit, page 2-3 (FANSIM)

          ! USE STATEMENTS:
          ! na

  IMPLICIT NONE    ! Enforce explicit typing of all variables in this routine

          ! SUBROUTINE ARGUMENT DEFINITIONS:
   Integer, Intent(IN) :: FanNum

          ! SUBROUTINE PARAMETER DEFINITIONS:
          ! na

          ! INTERFACE BLOCK SPECIFICATIONS
          ! na

          ! DERIVED TYPE DEFINITIONS
          ! na

          ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
      Real, External :: Psythw   ! calculate temperature from enthalpy and humidity ratio
      Real, External :: rhoairfn
      Real RhoAir
      Real DeltaPress  ! [N/M^2]
      Real FanEff
      Real MassFlow    ! [kg/sec]
      Real Tin         ! [C]
      Real Win
      Real PartLoadFrac
      REAL MaxFlowFrac   !Variable Volume Fan Max Flow Fraction [kg/sec]
      REAL MinFlowFrac   !Variable Volume Fan Min Flow Fraction [kg/sec]
      REAL FlowFrac      !Variable Volume Fan Flow Fraction [kg/sec]
      Real FanShaftPower ! power delivered to fan shaft
      Real PowerLossToAir ! fan and motor loss to air stream (watts)

! Simple Variable Volume Fan
! Type of Fan          Coeff1       Coeff2       Coeff3        Coeff4      Coeff5
! INLET VANE DAMPERS   0.35071223   0.30850535   -0.54137364   0.87198823  0.000
! DISCHARGE DAMPERS    0.37073425   0.97250253   -0.34240761   0.000       0.000
! VARIABLE SPEED MOTOR 0.0015302446 0.0052080574  1.1086242   -0.11635563  0.000

   DeltaPress = Fan(FanNum)%DeltaPress
   FanEff     = Fan(FanNum)%FanEff

   Tin        = Fan(FanNum)%InletAirTemp
   Win        = Fan(FanNum)%InletAirHumRat
   RhoAir     = RhoAirFn(OutBaroPress,Tin,Win)
   MassFlow   = MIN(Fan(FanNum)%InletAirMassFlowRate,Fan(FanNum)%MaxAirMassFlowRate)
   MassFlow   = MAX(MassFlow,Fan(FanNum)%MinAirMassFlowRate)

  ! Calculate and check limits on fraction of system flow
  MaxFlowFrac = 1.0
  ! MinFlowFrac is calculated from the ration of the volume flows and is non-dimensional
  MinFlowFrac = Fan(FanNum)%MinAirFlowRate/Fan(FanNum)%MaxAirFlowRate
  ! The actual flow fraction is calculated from MassFlow and the MaxVolumeFlow * AirDensity
  FlowFrac = MassFlow/(Fan(FanNum)%MaxAirMassFlowRate)
            !

! Calculate the part Load Fraction
  If(FlowFrac .gt. MaxFlowFrac) Then
     PartLoadFrac = 1.0
  Else If(FlowFrac .lt. MinFlowFrac) Then
     PartLoadFrac = MinFlowFrac
  Else
     PartLoadFrac=Fan(FanNum)%FanCoeff(1) + Fan(FanNum)%FanCoeff(2)*FlowFrac +  &
                  Fan(FanNum)%FanCoeff(3)*FlowFrac**2 + Fan(FanNum)%FanCoeff(4)*FlowFrac**3 + &
                  Fan(FanNum)%FanCoeff(5)*FlowFrac**4
  End If



   !Determine the Fan Schedule for the Time step
  If((GetCurrentScheduleValue(Fan(FanNum)%SchedPtr) .gt. 0.0) .and. &
        (Massflow .gt. 0.0)) Then
   !Fan is operating
   Fan(FanNum)%FanPower = PartLoadFrac*MassFlow*DeltaPress/(FanEff*RhoAir) ! total fan power
   FanShaftPower = Fan(FanNum)%MotEff * Fan(FanNum)%FanPower  ! power delivered to shaft
   PowerLossToAir = FanShaftPower + (Fan(FanNum)%FanPower - FanShaftPower) * &
                    Fan(FanNum)%MotInAirFrac
   Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy + PowerLossToAir/MassFlow
   ! This fan does not change the moisture or Mass Flow across the component
   Fan(FanNum)%OutletAirHumRat       = Fan(FanNum)%InletAirHumRat
   Fan(FanNum)%OutletAirMassFlowRate = MassFlow
   Fan(FanNum)%OutletAirTemp = Psythw(Fan(FanNum)%OutletAirEnthalpy,Fan(FanNum)%OutletAirHumRat)
   ! Set fan control information
   !Fan(FanNum)%MassFlowRateMaxAvail = MassFlow
   !Fan(FanNum)%MassFlowRateMinAvail = MassFlow
  Else
   !Fan is off and not operating no power consumed and mass flow rate.
   Fan(FanNum)%FanPower = 0.0
   FanShaftPower = 0.0
   PowerLossToAir = 0.0
   Fan(FanNum)%OutletAirMassFlowRate = 0.0
   Fan(FanNum)%OutletAirHumRat       = Fan(FanNum)%InletAirHumRat
   Fan(FanNum)%OutletAirEnthalpy     = Fan(FanNum)%InletAirEnthalpy
   Fan(FanNum)%OutletAirTemp = Fan(FanNum)%InletAirTemp
   ! Set the Control Flow variables to 0.0 flow when OFF.
   Fan(FanNum)%MassFlowRateMaxAvail = 0.0
   Fan(FanNum)%MassFlowRateMinAvail = 0.0
  End If

  RETURN
END SUBROUTINE SimVariableVolumeFan
```

```
SUBROUTINE SimZoneExhaustFan(FanNum)

          ! SUBROUTINE INFORMATION:
          !       AUTHOR         Fred Buhl
          !       DATE WRITTEN   Jan 2000
          !       MODIFIED       na
          !       RE-ENGINEERED  na

          ! PURPOSE OF THIS SUBROUTINE:
          ! This subroutine simulates the Zone Exhaust Fan

          ! METHODOLOGY EMPLOYED:
          ! Converts design pressure rise and efficiency into fan power and temperature rise
          ! Constant fan pressure rise is assumed.

          ! REFERENCES:
          ! ASHRAE HVAC 2 Toolkit, page 2-3 (FANSIM)

          ! USE STATEMENTS:
          ! na

  IMPLICIT NONE    ! Enforce explicit typing of all variables in this routine

          ! SUBROUTINE ARGUMENT DEFINITIONS:
   Integer, Intent(IN) :: FanNum

          ! SUBROUTINE PARAMETER DEFINITIONS:
          ! na

          ! INTERFACE BLOCK SPECIFICATIONS
          ! na

          ! DERIVED TYPE DEFINITIONS
          ! na

          ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
      Real RhoAir
      Real DeltaPress  ! [N/M^2]
      Real FanEff
      Real MassFlow    ! [kg/sec]
      Real Tin         ! [C]
      Real Win
      Real PowerLossToAir ! fan and motor loss to air stream (watts)
      Real, External :: Psythw   ! calculate temperature from enthalpy and humidity ratio
      Real, External :: rhoairfn

   DeltaPress = Fan(FanNum)%DeltaPress
   FanEff     = Fan(FanNum)%FanEff

   ! For a Constant Volume Simple Fan the Max Flow Rate is the Flow Rate for the fan
   Tin        = Fan(FanNum)%InletAirTemp
   Win        = Fan(FanNum)%InletAirHumRat
   RhoAir     = RhoAirFn(OutBaroPress,Tin,Win)
   MassFlow   = Fan(FanNum)%InletAirMassFlowRate
   !
   !Determine the Fan Schedule for the Time step
 If(GetCurrentScheduleValue(Fan(FanNum)%SchedPtr) .gt. 0.0) THEN
   !Fan is operating
   Fan(FanNum)%FanPower = MassFlow*DeltaPress/(FanEff*RhoAir) ! total fan power
   PowerLossToAir = Fan(FanNum)%FanPower
   Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy + PowerLossToAir/MassFlow
   ! This fan does not change the moisture or Mass Flow across the component
   Fan(FanNum)%OutletAirHumRat       = Fan(FanNum)%InletAirHumRat
   Fan(FanNum)%OutletAirMassFlowRate = MassFlow
   Fan(FanNum)%OutletAirTemp = Psythw(Fan(FanNum)%OutletAirEnthalpy,Fan(FanNum)%OutletAirHumRat)

 Else
   !Fan is off and not operating no power consumed and mass flow rate.
   Fan(FanNum)%FanPower = 0.0
   PowerLossToAir = 0.0
   Fan(FanNum)%OutletAirMassFlowRate = 0.0
   Fan(FanNum)%OutletAirHumRat       = Fan(FanNum)%InletAirHumRat
   Fan(FanNum)%OutletAirEnthalpy     = Fan(FanNum)%InletAirEnthalpy
   Fan(FanNum)%OutletAirTemp = Fan(FanNum)%InletAirTemp
   ! Set the Control Flow variables to 0.0 flow when OFF.
   Fan(FanNum)%MassFlowRateMaxAvail = 0.0
   Fan(FanNum)%MassFlowRateMinAvail = 0.0
   Fan(FanNum)%InletAirMassFlowRate = 0.0

 End If

 RETURN
END SUBROUTINE SimZoneExhaustFan

! End Algorithm Section of the Module
! ******************************************************************************

! Beginning of Update subroutines for the Fan Module
! ******************************************************************************

SUBROUTINE UpdateFan(FanNum)

          ! SUBROUTINE INFORMATION:
          !       AUTHOR         Richard Liesen
          !       DATE WRITTEN   April 1998
          !       MODIFIED       na
          !       RE-ENGINEERED  na

          ! PURPOSE OF THIS SUBROUTINE:
          ! This subroutine updates the fan outlet nodes.

          ! METHODOLOGY EMPLOYED:
          ! Data is moved from the fan data structure to the fan outlet nodes.

          ! REFERENCES:
          ! na

          ! USE STATEMENTS:
          ! na

  IMPLICIT NONE    ! Enforce explicit typing of all variables in this routine

          ! SUBROUTINE ARGUMENT DEFINITIONS:
   Integer, Intent(IN) :: FanNum

          ! SUBROUTINE PARAMETER DEFINITIONS:
          ! na

          ! INTERFACE BLOCK SPECIFICATIONS
          ! na

          ! DERIVED TYPE DEFINITIONS
          ! na

          ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
  Integer             :: OutletNode
  Integer             :: InletNode


   OutletNode = Fan(FanNum)%OutletNodeNum
   InletNode = Fan(FanNum)%InletNodeNum

   ! Set the outlet air nodes of the fan
   Node(OutletNode)%MassFlowRate  = Fan(FanNum)%OutletAirMassFlowRate
   Node(OutletNode)%Temp          = Fan(FanNum)%OutletAirTemp
   Node(OutletNode)%HumRat        = Fan(FanNum)%OutletAirHumRat
   Node(OutletNode)%Enthalpy      = Fan(FanNum)%OutletAirEnthalpy
   ! Set the outlet nodes for properties that just pass through & not used
   Node(OutletNode)%Quality         = Node(InletNode)%Quality
   Node(OutletNode)%Press           = Node(InletNode)%Press
   ! Set the Node Flow Control Variables from the Fan Control Variables
   Node(OutletNode)%MassFlowRateMaxAvail = Fan(FanNum)%MassFlowRateMaxAvail
   Node(OutletNode)%MassFlowRateMinAvail = Fan(FanNum)%MassFlowRateMinAvail

   IF (Fan(FanNum)%FanType .EQ. 'ZONE EXHAUST FAN') THEN
     Node(InletNode)%MassFlowRate = Fan(FanNum)%InletAirMassFlowRate
   END IF

  RETURN
END Subroutine UpdateFan

!        End of Update subroutines for the Fan Module
! *****************************************************************************


! Beginning of Reporting subroutines for the Fan Module
! *****************************************************************************

SUBROUTINE ReportFan(FanNum)

          ! SUBROUTINE INFORMATION:
          !       AUTHOR         Richard Liesen
          !       DATE WRITTEN   April 1998
          !       MODIFIED       na
          !       RE-ENGINEERED  na

          ! PURPOSE OF THIS SUBROUTINE:
          ! This subroutine updates the report variables for the fans.

          ! METHODOLOGY EMPLOYED:
          ! na

          ! REFERENCES:
          ! na

          ! USE STATEMENTS:
  Use DataHVACGlobals, ONLY: TimeStepSys, FanElecPower

  IMPLICIT NONE    ! Enforce explicit typing of all variables in this routine

          ! SUBROUTINE ARGUMENT DEFINITIONS:
   Integer, Intent(IN) :: FanNum

          ! SUBROUTINE PARAMETER DEFINITIONS:
          ! na

          ! INTERFACE BLOCK SPECIFICATIONS
          ! na

          ! DERIVED TYPE DEFINITIONS
          ! na

          ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
          ! na

    Fan(FanNum)%FanEnergy=Fan(FanNum)%FanPower*TimeStepSys*3600
    Fan(FanNum)%DeltaTemp=Fan(FanNum)%OutletAirTemp - Fan(FanNum)%InletAirTemp
    FanElecPower = Fan(FanNum)%FanPower

  RETURN
END Subroutine ReportFan

!        End of Reporting subroutines for the Fan Module
! *****************************************************************************

!                                  COPYRIGHT NOTICE
!
!     Copyright © 1996-2001 The Board of Trustees of the University of Illinois
!     and The Regents of the University of California, pending approval by the
!     US Department of Energy.  All rights reserved.
!
```

```
!     NOTICE: The U.S. Government is granted for itself and others acting on its
!     behalf a paid-up, nonexclusive, irrevocable, worldwide license in this data to
!     reproduce, prepare derivative works, and perform publicly and display publicly.
!     Beginning five (5) years after permission to assert copyright is granted,
!     subject to two possible five year renewals, the U.S. Government is granted for
!     itself and others acting on its behalf a paid-up, non-exclusive, irrevocable
!     worldwide license in this data to reproduce, prepare derivative works,
!     distribute copies to the public, perform publicly and display publicly, and to
!     permit others to do so.
!
!     TRADEMARKS: DOE-2.1e, DOE-2, and DOE are trademarks of the US Department of
!     Energy.
!
!     DISCLAIMER OF LIABILITY: With respect to this software, neither UI, LBNL, the
!     Department of Energy, the US Government and/or any person or organization acting
!     on behalf of any of them:
!
!     a.  Make any warranty or representation whatsoever, express or implied, with
!     respect to EnergyPlus or any derivative works thereof, including without
!     limitation warranties of merchantability, warranties of fitness for a particular
!     purpose, or warranties or representations regarding the use, or the results of
!     the use of EnergyPlus or derivative works thereof in terms of correctness,
!     accuracy, reliability, currentness, or otherwise.  The entire risk as to the
!     results and performance of the licensed software is assumed by the licensee.
!
!     b.  Make any representation or warranty that EnergyPlus or derivative works
!     thereof will not infringe any copyright or other proprietary right.
!
!     c.  Assume any liability whatsoever with respect to any use of EnergyPlus,
!     derivative works thereof, or any portion thereof or with respect to any damages
!     which may result from such use.
!
!     DISCLAIMER OF ENDORSEMENT: Reference herein to any specific commercial products,
!     process, or service by trade name, trademark, manufacturer, or otherwise, does
!     not necessarily constitute or imply its endorsement, recommendation, or favoring
!     by the United States Government, the University of California, the University of
!     Illinois or the Ernest Orlando Lawrence Berkeley National Laboratory.
!
!

End Module Fans
```

This example can be used as a template for new HVAC component modules. In particular, the commenting structure in the module and within the subroutines should be followed closely. Of course, there is no perfect example module – this one is particularly simple. Some others that might be examined are in files Humidifiers.f90, HVACHeatingCoils.f90 and PlantChillers.f90. Templates are also available as separate files in *Internal Documentation / Programming Templates*.

How it fits together

Although we have designed the EnergyPlus modules to be as independent as possible, obviously they cannot be completely independent. How does an EnergyPlus HVAC module fit in with the rest of the program? First, we will show some subroutine calling trees that will display the overall program structure.

Top Level Calling Tree

EnergyPlus

- ProcessInput (in InputProcessor)
- ManageSimulation (in SimulationManager)
  - ManageWeather (in WeatherManager)
  - ManageHeatBalance (in HeatBalanceManager)
    - ManageSurfaceHeatBalance (in HeatBalanceSurfaceManager)
    - ManageAirHeatBalance (in HeatBalanceAirManager)
      - CalcHeatBalanceAir (in HeatBalanceAirManager)
        - ManageHVAC (in HVACManager)

The HVAC part of EnergyPlus is divided into a number of simulation blocks. At this point, there are blocks for the air system, the zone equipment, the plant supply, the plant demand, the condenser supply, and the condenser demand. There will be simulation blocks for waste heat supply and usage as well as electricity and gas. Within each HVAC time step, the blocks are simulated repeatedly until the conditions on each side of each block interface match up. The following calling tree represents the high level HVAC simulation structure. It is schematic – not all routines are shown.

High Level HVAC Calling Tree

ManageHVAC (in HVACManager)

- ZoneAirUpdate('PREDICT', . . .) (in HVACManager)
  estimate the zone heating or cooling demand

- SimHVAC (in HVACManager)
  - ManageSetPoints (in SetPointManager)
  - SimSelectedEquipment (in HVACManager)
    - ManageAirLoops (in SimAirServingZones)
    - ManageZoneEquipment (in ZoneEquipmentManager)
    - ManagePlantSupplySides (in PlantLoopSupplySideManager)
    - ManagePlantDemandSides (in PlantdemandSideLoops)
    - ManageCondSupplySides (in CondLoopManager)
    - ManageCondenserDemandSides (in CondenserDemandSideLoops)

- ZoneAirUpdate('CORRECT', . . .) (in HVACManager)
  From the amount of heating and cooling actually provided by the HVAC system, calculate the zones' temperatures.

Each of the "Manage" routines has a different structure, since the simulation to be performed is different in each case. We will show schematic calling trees for several of the "Manage" routines.

Air System Calling Tree

ManageAirLoops (in SimAirServingZones)

- GetAirPathData (in SimAirServingZones)
- InitAirLoops (in SimAirServingZones)
- SimAirLoops (in SimAirServingZones)
  - SimAirLoopComponent (in SimAirServingZones)
  - UpdateBranchConnections (in SimAirServingZones)
    - ManageOutsideAirSystem (in MixedAir)
      - SimOutsideAirSys (in MixedAir)
        - SimOAController (in MixedAir)
        - SimOAComponent (in Mixed Air)
          - SimOAMixer (in MixedAir)
          - SimulateFanComponents(in FanSimulation; file HVACFanComponent)
          - SimulateWaterCoilComponents (in WaterCoilSimulation; file HVACWaterCoilComponent)
          - SimHeatRecovery (in HeatRecovery)
          - SimDesiccantDehumidifier (in DesiccantDehumidifiers)
    - SimulateFanComponents (in FanSimulation; file HVACFanComponent)
    - SimulateWaterCoilComponents (in WaterCoilSimulation; file HVACWaterCoilComponent)
    - SimulateHeatingCoilComponents (in HeatingCoils; file HVACHeatingCoils)
    - SimDXCoolingSystem (in HVACDXSystem)
    - SimFurnace (in Furnaces; file HVACFurnace)
    - SimHumidifier (in Humidifiers)
    - SimEvapCooler (in EvaporativeCoolers; file HVACEvapComponent)
    - SimDesiccantDehumidifier (in DesiccantDehumidifiers)
    - SimHeatRecovery (in HeatRecovery)
  - ManageControllers (in Controllers)
    - GetControllerInput (in Controllers)
    - InitController (in Controllers)
    - SimpleController (in Controllers)
    - LimitController (in Controllers)
    - UpdateController (in Controllers)
    - Report Controller (in Controllers)
  - ResolveSysFlow (in SimAirServingZones)
  - UpdateHVACInterface (in HVACInterfaceManager)
- ReportAirLoops (in SimAirServingZones)

Plant Supply Calling Tree

ManagePlantSupplySides (in PlantLoopSupplySideManager)

- GetLoopData (in PlantLoopSupplySideManager)
- SetLoopInitialConditions (in PlantLoopSupplySideManager)

- CalcLoopDemand (in PlantLoopSupplySideManager)
- ManagePlantLoopOperation (in PlantCondLoopOperation)
- DistributeLoad (in PlantLoopSupplySideManager)
- SimPlantEquip (in PlantLoopSupplySideManager)
  - SimPipes (in Pipes; file PlantPipes)
  - SimPumps (in Pumps; file PlantPumps)
  - SimEngineDrivenChiller (in ChillerEngineDriven ; file PlantChillers)
  - SimBLASTAbsorber (in ChillerAbsorption ; file PlantAbsorptionChillers)
  - SimElectricChiller (in ChillerElectric ; file PlantChillers)
  - SimGTChiller (in ChillerGasTurbine ; file PlantChillers)
  - SimConstCOPChiller (in ChillerConstCOP; file PlantChillers)
  - SimBLASTChiller (in ChillerBLAST ; file PlantChillers)
  - SimOutsideCooling (in OutsideCoolingSources ; file PlantOutsideCoolingSources)
  - SimGasAbsorber (in ChillerGasAbsorption ; file PlantGasAbsorptionChiller)
  - SimBoiler (in Boilers; file PlantBoilers)
  - SimWaterHeater (in WaterHeaters ; file PlantWaterHeater)
  - SimOutsideHeating (in OutsideHeatingSources; file PlantOutsideHeatingSources)
- UpdateSplitter (in PlantLoopSupplySideManager)
- SolveFlowNetwork (in PlantLoopSupplySideManager)
- CalcLoopDemand (in PlantLoopSupplySideManager)
- SimPlantEquip (in PlantLoopSupplySideManager)
- UpdateSplitter
- UpdateMixer (in PlantLoopSupplySideManager)
- SimPlantEquip (in PlantLoopSupplySideManager)
- CheckLoopExitNodes (in PlantLoopSupplySideManager)
- UpdateHVACInterface (in HVACInterfaceManager)
- UpdateReportVars (in PlantLoopSupplySideManager)

Zone Equipment Calling Tree

ManageZoneEquipment (in ZoneEquipmentManager)

- GetZoneEquipment (in ZoneEquipmentManager)
- InitZoneEquipment (in ZoneEquipmentManager)
- SimZoneEquioment (in ZoneEquipmentManager)
  - SimAirLoopSplitter (in Splitters; file HVACSplitterComponent)
  - SimAirZonePlenum (in ZonePlenum; file ZonePlenumComponent)
  - SetZoneEquipSimOrder (in ZoneEquipmentManager)
  - InitSystemOutputRequired (in ZoneEquipmentManager)
  - ManageZoneAirLoopEquipment (in ZoneAirLoopEquipmentManager)
    - GetZoneAirLoopEquipment (in ZoneAirLoopEquipmentManager)
    - SimZoneAirLoopEquipment (in ZoneAirLoopEquipmentManager)
      - SimulateDualDuct (in DualDuct; file HVACDualDuctSystem)
        - GetDualDuctInput (in DualDuct; file HVACDualDuctSystem)
        - InitDualDuct (in DualDuct; file HVACDualDuctSystem)
        - SimDualDuctConstVol (in DualDuct; file HVACDualDuctSystem)
        - SimDualDuctVarVol (in DualDuct; file HVACDualDuctSystem)

            - UpdateDualDuct (in DualDuct; file HVACDualDuctSystem)
            - ReportDualDuct (in DualDuct; file HVACDualDuctSystem)
        - SimulateSingleDuct (in SingleDuct; file HVACSingleDuctSystem)
            - GetSysInput (in SingleDuct; file HVACSingleDuctSystem)
            - InitSys (in SingleDuct; file HVACSingleDuctSystem)
            - SimConstVol (in SingleDuct; file HVACSingleDuctSystem)
            - SimVAV (in SingleDuct; file HVACSingleDuctSystem)
            - ReportSys (in SingleDuct; file HVACSingleDuctSystem)
        - SimPIU (in PoweredInductionUnits)
            - GetPIUs (in PoweredInductionUnits)
            - InitPIUs (in PoweredInductionUnits)
            - CalcSeriesPIU (in PoweredInductionUnits)
            - CalcParallelPIU (in PoweredInductionUnits)
            - ReportPIU (in PoweredInductionUnits)
    - SimDirectAir (in DirectAirManager; file DirectAir)
    - SimPurchasedAir (in PurchasedAirManager)
    - SimWindowAC (in WindowAC)
    - SimFanCoilUnit (in FanCoilUnits)
    - SimUnitVentilator (in UnitVentilator)
    - SimUnitHeater (in UnitHeater)
    - SimBaseboard (in BaseboardRadiator)
    - SimHighTempRadiantSystem (in HighTempRadiantSystem; file RadiantSystemHighTemp)
    - SimLowTempRadiantSystem (in LowTempRadiantSystem; file RadiantSystemLowTemp)
    - SimulateFanComponents (in Fans; file HVACFanComponent)
    - SimHeatRecovery (in HeatRecovery)
    - UpdateSystemOutputRequired (in ZoneEquipmentManager)
    - SimAirLoopSplitter (in Splitters; file HVACSplitterComponent)
    - SimAirZonePlenum (in ZonePlenum; file ZonePlenumComponent)
    - CalcZoneMassBalance (in ZoneEquipmentManager)
    - CalcZoneLeavingConditions (in ZoneEquipmentManager)
    - SimReturnAirPath (in ReturnAirPathManager; file ReturnAirPath)
        - SimAirMixer (in Mixers; HVACMixerComponent)
        - SimAirZonePlenum (in ZonePlenum; file ZonePlenumComponent)
- RecordZoneEquipment (in ZoneEquipmentManager)
- ReportZoneEquipment (in ZoneEquipmentManager)

Inserting the New Module into the Program

Let us return to our example new module NewHVACComponent. Since the module does its own input and output, adding the NewHVACComponent model to the program simply means adding a call to the driver routine SimNewHVACComponent from the correct place in EnergyPlus. In the simplest case, there is only one location from which the driver routine should be called. In some cases, though, more than one HVAC simulation block will need to use the new component model. SimulateWaterCoilComponents, for instance, is used in the zone equipment for reheat coils, and in the air system for heating and cooling coils. In the air system simulation it is called from two places: the main air system simulation, and the mixed air simulation – the outside air duct might contain a separate cooling coil.

Let us assume that the NewHVACComponent will be part of the air system – perhaps it is a solid desiccant wheel. Examining the air system calling tree we see that SimAirLoopComponent is one routine that will invoke the new component, and - if we want the component to possibly be in the outside air stream – then SimOAComponent is the other routine that will need to call the new component simulation. Generally, all that is involved is adding a new CASE statement to a Fortran SELECT construct. For instance in SimAirLoopComponent this would look like:

```
SELECT CASE(CompType)

CASE('OUTSIDE AIR SYSTEM')
    CALL ManageOutsideAirSystem(CompName,FirstHVACIteration,LastSim)

! Fan Types for the air sys simulation
  CASE('FAN:SIMPLE:CONSTVOLUME')
    CALL SimulateFanComponents(CompName,FirstHVACIteration)
  CASE('FAN:SIMPLE:VARIABLEVOLUME')
    CALL SimulateFanComponents(CompName,FirstHVACIteration)

! Coil Types for the air sys simulation
  CASE('COIL:WATER:SIMPLECOOLING')
    CALL SimulateWaterCoilComponents(CompName,FirstHVACIteration)
  CASE('COIL:WATER:SIMPLEHEATING')
    CALL SimulateWaterCoilComponents(CompName,FirstHVACIteration)
  CASE('COIL:WATER:DETAILEDFLATCOOLING')
    CALL SimulateWaterCoilComponents(CompName,FirstHVACIteration)
  CASE('COIL:ELECTRIC:HEATING')
    CALL SimulateHeatingCoilComponents(CompName=CompName,&
           FirstHVACIteration=FirstHVACIteration, &
           QCoilReq=0.0)
CASE('COIL:GAS:HEATING')
    CALL SimulateHeatingCoilComponents(CompName=CompName,&
           FirstHVACIteration=FirstHVACIteration, &
           QCoilReq=0.0)
  CASE('DXSYSTEM:AIRLOOP')
         CALL SimDXCoolingSystem(CompName)

  CASE('FURNACE:BLOWTHRU:HEATONLY')
         CALL SimFurnace(CompName, FirstHVACIteration)
  CASE('FURNACE:BLOWTHRU:HEATCOOL')
         CALL SimFurnace(CompName, FirstHVACIteration)
  CASE('UNITARYSYSTEM:BLOWTHRU:HEATONLY')
         CALL SimFurnace(CompName, FirstHVACIteration)
  CASE('UNITARYSYSTEM:BLOWTHRU:HEATCOOL')
         CALL SimFurnace(CompName, FirstHVACIteration)

! Humidifier Types for the air system simulation
  CASE('HUMIDIFIER:STEAM:ELECTRICAL')
    CALL SimHumidifier(CompName,FirstHVACIteration)

! Evap Cooler Types for the air system simulation
  CASE('EVAPCOOLER:DIRECT:CELDEKPAD')
    CALL SimEvapCooler(CompName)
  CASE('EVAPCOOLER:INDIRECT:CELDEKPAD')
    CALL SimEvapCooler(CompName)
  CASE('EVAPCOOLER:INDIRECT:WETCOIL')
    CALL SimEvapCooler(CompName)

! Desiccant Dehumidifier Types for the air system simulation
  CASE('DESICCANT DEHUMIDIFIER:SOLID')
    CALL SimDesiccantDehumidifier(CompName,FirstHVACIteration)
```

```
! Heat recovery
  CASE('HEAT EXCHANGER:AIR TO AIR:FLAT PLATE')
    CALL SimHeatRecovery(CompName,FirstHVACIteration)

! New HVAC Component
  CASE ('NEW HVAC COMPONENT')
    CALL SimNewHVACComponent(CompName,FirstHVACIteration)

END SELECT
```

The new code is italicized. Do the same thing in SimOAComponent and you are done! Note that “NEW HVAC COMPONENT” is the class name (keyword) for the new component on the IDD file. The class names are converted to upper case in EnergyPlus, so the CASE statement must have the class name in upper case. The actual class name on the IDD file would probably be “New HVAC Component.

If the new HVAC component is a piece of zone equipment – a cooled beam system, for instance – then the zone equipment calling tree indicates that the call to SimNewHVACComponent would be in SimZoneEquipment. If the new component is a gas fired absorption chiller, the call would be in SimPlantEquip.

In every case, since NewHVACComponent is a new module, a USE statement must be added to the calling subroutine. For instance in SimAirLoopComponent this would look like:

```
SUBROUTINE SimAirLoopComponent(CompType, CompName, FirstHVACIteration,
LastSim)

          ! SUBROUTINE INFORMATION
          !             AUTHOR:  Russ Taylor, Dan Fisher, Fred Buhl
          !       DATE WRITTEN:  Oct 1997
          !           MODIFIED:  Dec 1997 Fred Buhl
          !      RE-ENGINEERED:  This is new code, not reengineered

          ! PURPOSE OF THIS SUBROUTINE:
          ! Calls the individual air loop component simulation
routines

          ! METHODOLOGY EMPLOYED: None

          ! REFERENCES: None

          ! USE Statements
  USE Fans,       Only:SimulateFanComponents
  USE WaterCoils, Only:SimulateWaterCoilComponents
  USE MixedAir,   Only:ManageOutsideAirSystem
  USE NewHVACComponent,    Only:SimNewHVACComponent
```

Considerations for Legacy Codes

Some special considerations should be taken by those module developers who are adding to EnergyPlus’s capabilities by adapting existing codes into the module structure.

Legacy codes will typically come with their own input and output structures. In adapting this to use with EnergyPlus, the module developer will usually want to bypass these routines by either embedding the code into EnergyPlus and using input entirely from the IDD/IDF structure OR writing a simple input file to the legacy code. Either of these approaches can be used.

Code Readability vs. Speed of Execution

Programmers throughout time have had to deal with speed of code execution and it's an ongoing concern. However, compilers are pretty smart these days and, often, can produce speedier code for the hardware platform than the programmer can when he or she uses "speed up" tips. The EnergyPlus development team would rather the code be more "readable" to all than to try to outwit the compilers for every platform. First and foremost, the code is the true document of what EnergyPlus does – other documents will try to explain algorithms and such but must really take a back seat to the code itself.

However, many people may read the code – as developers, we should try to make it as readable at first glance as possible. For a true example from the code and a general indication of preferred style, take the case of the zone temperature update equation. In the engineering document, the form is recognizable and usual:

$$T_z^t = \frac{\sum_{i=1}^{N_{sl}} \dot{Q}_i + \sum_{i=1}^{N_{surfaces}} h_i A_i T_{si} + \sum_{i=1}^{N_{zones}} \dot{m}_i C_p T_{zi} + \dot{m}_{\inf} C_p T_\infty + \dot{m}_{sys} C_p T_{\text{supply}} - \left(\frac{C_z}{\delta t}\right)\left(-3T_z^{t-\delta t} + \frac{3}{2}T_z^{t-2\delta t} - \frac{1}{3}T_z^{t-3\delta t}\right)}{\left(\frac{11}{6}\right)\frac{C_z}{\delta t} + \sum_{i=1}^{N_{surfaces}} h_i A + \sum_{i=1}^{N_{zones}} \dot{m}_i C_p + \dot{m}_{\inf} C_p + \dot{m}_{sys} C}$$

And, this equation appears in the code (ZoneTempPredictorCorrector Module), as:

```
ZT(ZoneNum)= (CoefSumhat +   CoefAirrat*(3.0*ZTM1(ZoneNum) - (3.0/2.0)*ZTM2(ZoneNum) &
                                  + (1./3.)* ZTM3(ZoneNum))) &
                             / ((11.0/6.0)*CoefAirrat+CoefSumha)
```

somewhat abbreviated here due to lack of page width but still recognizable from the original. A better version would actually be:

```
ZT(ZoneNum)= (CoefSumhat -   CoefAirrat*(-3.0*ZTM1(ZoneNum) + (3.0/2.0)*ZTM2(ZoneNum) &
                                  - (1./3.)* ZTM3(ZoneNum))) &
                             / ((11.0/6.0)*CoefAirrat+CoefSumha)
```

whereas the natural tendency of programming would lead to the less readable:

```
ZT(ZoneNum)= (CoefSumhat + CoefAirrat*(3.0*ZTM1(ZoneNum) - 1.5*ZTM2(ZoneNum) + .333333*
ZTM3(ZoneNum))) &
           / (1.83333*CoefAirrat+CoefSumha)
```

The final version is a correct translation (more or less) from the Engineering/usual representation but much harder to look at in code and realize what is being represented.

EnergyPlus Services

EnergyPlus provides some standard services that make the developers task much easier. The developer can concentrate on the new simulation algorithm rather than have to deal with details of input file structure, writing output, obtaining scheduled data, and accessing weather variables.

Global Data

We have tried to limit the amount of global data in EnergyPlus development. Two critical data-only modules, however, have been used:

DataGlobals – contains truly global data (such as number of zones, current hour, simulation status flags, interface statements to error and output routines)

DataEnvironment – contains weather data that is global (current dry bulb, outside barometric pressure)

These are used in routines as the examples have illustrated.

Input Services

All input is processed by the EnergyPlus module: *InputProcessor*. The InputProcessor uses the definition lines in the IDD as directives on how to process each input object. The InputProcessor also turns all alpha strings into all UPPER CASE. Currently, it does nothing else to those strings – so the number of blanks in a string must match what the calculational modules expect. The InputProcessor processes all numeric strings into single precision real numbers. Special characters, such as tabs, should *not* be included in the IDF.

The EnergyPlus module *InputProcessor* provides several routines - generically called the “get” routines – that enable the developer to readily access the data for a new module. These routines are made available by including a “USE InputProcessor” statement in the module or in the routine that will use the “get” routines. The GetFanInput subroutine in the example illustrates some of the uses of the “get” routines.

GetNumObjectsFound

This function returns the number of objects in the input belonging to a particular class. In other terms, it returns the number of instances in the input of a particular component.

```
Example:
NumVAVSys = GetNumObjectsFound('SINGLE DUCT:VAV:REHEAT')
```

Here NumVAVSys will contain the number of single duct VAV terminal units in the input. SINGLE DUCT:VAV:REHEAT is the class name or keyword defining VAV terminal unit input on the IDD file.

GetObjectItem

This subroutine is used to obtain the actual alphanumeric and numeric data for a particular object.

Example:

```
INTEGER :: SysNum
INTEGER :: SysIndex
INTEGER :: NumAlphas
INTEGER :: NumNums
INTEGER :: IOSTAT
REAL, DIMENSION(5) :: NumArray
CHARACTER(len=MaxNameLength), DIMENSION(8) :: AlphArray
. . . . . . . . .
! Flow
NumVAVSys = GetNumObjectsFound('SINGLE DUCT:VAV:REHEAT')
. . . . . . . . .
!Start Loading the System Input
DO SysIndex = 1, NumVAVSys
  CALL GetObjectItem('SINGLE DUCT:VAV:REHEAT',SysIndex,AlphArray,&
                      NumAlphas,NumArray,NumNums,IOSTAT)
  SysNum = SysIndex
  . . . . . . . .
  Sys(SysNum)%SysName     = AlphArray(1)
  Sys(SysNum)%SysType     = 'SINGLE DUCT:VAV:REHEAT'
  Sys(SysNum)%ReheatComp     = AlphArray(6)
  Sys(SysNum)%ReheatName     = AlphArray(7)
  . . . . . . . .
END DO   ! end Number of Sys Loop    END IF
```

Here GetObjectItem is called with inputs 'SINGLE DUCT:VAV:REHEAT' – the class of object we want to input – and SysIndex – the index of the object on the input file. If SysIndex is 3, the call to GetObjectItem will get the data for the third VAV terminal unit on the input file. Output is returned in the remaining arguments. AlphArray contains in order all the alphanumeric data items for a single VAV terminal unit. NumArray contains all the numeric data items. NumAlphas is the number of alphanumeric items read; NumNums is the number of numeric data items read. IOSTAT is a status flag: -1 means there was an error; +1 means the input was OK. AlphArray and NumArray should be dimensioned to handle the largest expected input for the item.

GetNodeNums

Nodes have unique names on the input file. Internally nodes are referenced as numbers. GetNodeNums handles the assignment of a unique number to each node.

```
Example:
CHARACTER(len=MaxNameLength), DIMENSION(4) :: AlphArray
INTEGER :: NumNodes
INTEGER, DIMENSION(25) :: NodeNums
. . . . . . . .
CALL GetNodeNums(AlphArray(3),NumNodes,NodeNums)
. . . . . . . .
Fan(FanNum)%InletNodeNum  = NodeNums(1)
```

The first argument is a node name or the name of a Node List, the second argument is the number of nodes in the Node List (1 for a single node), and the last argument is the output: a list of node numbers.

GetObjectItemNum

GetObjectItem requires the input file index of the desired object in order to get the object's data. Sometimes this index may be unknown, but the name of the object is known. GetObjectItemNum returns the input file index given the class name and object name.

```
Example:
ListNum = GetObjectItemNum('CONTROLLER LIST',ControllerListName)
```

In the example, ListNum will contain the input file index of the 'CONTROLLER LIST' whose name is contained in the string variable ControllerListName.

FindItemInList

```
This function looks up a string in a similar list of items and returns
the index of the item in the list, if found. It is case sensitive.
Example:
SysNum = FindItemInList(CompName,Sys%SysName,NumSys)
```

CompName is the input string, Sys%SysName is the list of names to be searched, and NumSys is the size of the list.

A case insensitive version of the routine is called FindItem (same description).

SameString

This function returns true if two strings are equal (case insensitively).

```
Example:
IF (SameString(InputRoughness,'VeryRough')) THEN
    Material(MaterNum)%Roughness=VeryRough
ENDIF
```

VerifyName

This subroutine checks that an object name is unique; that is, it hasn't already been used for the same class of object and the name is not blank.

```
Example:
CALL VerifyName(AlphArray(1),Fan%FanName, &
  FanNum-1,IsNotOK,IsBlank,'FAN:SIMPLE:CONSTVOLUME Name')
```

The first argument is the name to be checked, the second is the list of names to search, the third argument is number of entries in the list, the 4th argument is set to TRUE is verification fails, the 5th argument is set to true if the name is blank, and the last argument is part of the error message written to the error file when verification fails.

Schedule Services

Schedules are widely used in specifying input for building simulation programs. For instance heat gains from lighting, equipment and occupancy are usually specified using schedules. Schedules are used to indicate when equipment is on or off. Schedules are also used to specify zone and system set points. EnergyPlus uses schedules in all these ways and provides services that make using schedules very easy for the developer.

Schedules are specified in a three level hierarchy in EnergyPlus input. The relevant portion of the IDD file is:

```
\group Schedules

ScheduleType,
       \memo ScheduleType contains ScheduleTypes for DAYSCHEDULE, and SCHEDULE
  A1,  \field ScheduleType Name
       \reference ScheduleTypeNames
       \memo used to validate ScheduleType in SCHEDULE and DAYSCHEDULE objects
  A2,  \field range
       \note put in minimum:maximum here or blank if not a limited value, e.g.
       \note 0.0:1.0 for fraction
  A3;  \field Numeric Type
       \note Numeric type is either Continuous (all numbers within the min and
       \note max are valid or Discrete (only integer numbers between min and
       \note max are valid.  (Could also allow REAL and INTEGER to mean the
       \note same things
       \type choice
       \key CONTINUOUS
       \key DISCRETE

DAYSCHEDULE,
       \memo A DAYSCHEDULE contains 24 values for each hour of the day.
  A1 , \field Name
       \type alpha
       \reference DayScheduleNames
  A2 , \field ScheduleType
       \type object-list
       \object-list ScheduleTypeNames
  N1 , \field Hour 1
       \type real
  N2 , \field Hour 2
       \type real
  . . . . . . . . . . . . . . .
  N24; \field Hour 24
       \type real

WEEKSCHEDULE,
       \memo  A WEEKSCHEDULE contains 12 DAYSCHEDULES, one for each day type.
  A1 , \field Name
       \reference WeekScheduleNames
       \type alpha
  A2 , \field Sunday
       \type object-list
       \object-list DayScheduleNames
       \note DAYSCHEDULE Name
  . . . . . . . . . . . . . . .
  A7 , \field Friday
       \type object-list
       \object-list DayScheduleNames
       \note DAYSCHEDULE Name
  A8 , \field Saturday
       \type object-list
       \object-list DayScheduleNames
       \note DAYSCHEDULE Name
  A9 , \field Holiday
       \type object-list
       \object-list DayScheduleNames
       \note DAYSCHEDULE Name
  A10, \field Summer Design Day
       \type object-list
       \object-list DayScheduleNames
       \note DAYSCHEDULE Name
  A11, \field Winter Design Day
       \type object-list
       \object-list DayScheduleNames
       \note DAYSCHEDULE Name
  A12, \field Special Day1
       \type object-list
       \object-list DayScheduleNames
       \note DAYSCHEDULE Name
  A13; \field Special Day2
       \type object-list
       \object-list DayScheduleNames
       \note DAYSCHEDULE Name

SCHEDULE,
       \memo A SCHEDULE contains from 1 to 18 WEEKSCHEDULES
  A1 , \field Name
       \type alpha
       \reference ScheduleNames
  A2 , \field ScheduleType
       \type object-list
       \object-list ScheduleTypeNames
  A3 , \field Name of WEEKSCHEDULE 1
       \type object-list
       \object-list WeekScheduleNames
  N1 , \field Start Month 1
       \type integer
       \minimum 1
       \maximum 12
  N2 , \field Start Day 1
       \type integer
       \minimum 1
       \maximum 31
  N3 , \field End Month 1
       \type integer
       \minimum 1
       \maximum 12
  N4 , \field End Day 1
       \type integer
       \minimum 1
       \maximum 31
  . . . . . . . . . . . . . .

  A20, \field Name of WEEKSCHEDULE 18
       \type object-list
       \object-list WeekScheduleNames
  N69, \field Start Month 18
       \type integer
       \minimum 1
       \maximum 12
  N70, \field Start Day 18
       \type integer
       \minimum 1
       \maximum 31
  N71, \field End Month 18
       \type integer
       \minimum 1
       \maximum 12
  N72; \field End Day 18
       \type integer
       \minimum 1
       \maximum 31
```

An example from in input (IDF) file:

```
ScheduleType,Fraction, 0.0 : 1.0 ,CONTINUOUS;
. . . . . . . .
    DAYSCHEDULE, Day On Peak, Fraction,
       0.,0.,0.,0.,0.,0.,0.,0.,0.,1.,1.,1.,1.,1.,1.,1.,1.,1.,0.,0.,0.,0.,0.,0.;
    WEEKSCHEDULE, Week on Peak,
        Day On Peak,Day On Peak,Day On Peak,
        Day On Peak,Day On Peak,Day On Peak,
        Day On Peak,Day On Peak,Day On Peak,
        Day On Peak,Day On Peak,Day On Peak;
    SCHEDULE, On Peak, Fraction,
        Week On Peak, 1,1, 12,31;
```

A day schedule assigns a number to each hour of the day. The week schedule assigns a day schedule to each day of the week plus holiday and some special days. Schedule assigns week schedules to various periods of the year. Both day schedules and schedules reference a schedule type. A schedule type is characterized by a range (e.g. 0 to 1) and whether it is continuous (can assume any value) or discrete (can assume integer values only). The following routines from the ScheduleManager module enable the developer to use schedules in a simulation.

GetScheduleIndex

This function takes a schedule name as input and returns an internal pointer to the schedule. Schedule values will always be accessed via the pointer not the name during the simulation for reasons of efficiency. This function should be called once for each schedule during the input phase and the returned value stored in the appropriate data structure.

```
Example
Baseboard(BaseboardNum)%SchedPtr = GetScheduleIndex(AlphArray(2))
```

Here the schedule pointer for the schedule name contained in AlphArray(2) is stored in the baseboard data structure for later use.

GetCurrentScheduleValue

This function returns the current schedule value for the current day and time, given the schedule pointer as input. An optional second argument can be used to specify a different hour than the current global hour of the day.

```
Example
CloUnit = GetCurrentScheduleValue(People(PeopleNum)%ClothingPtr)
```

Notice that the developer doesn’t have to keep track of hour of the day, day of the month, or month. The program does all of that. The only input needed is the pointer to the schedule.

Performance Curve Services

Some HVAC equipment models in EnergyPlus use performance curves. These are polynomials in one or two independent variables that are used to modify rated equipment performance for performance at the current, off-rated conditions. Most often the curves are functions of temperature – entering wetbulb and outside drybulb, for instance – or of the part load fraction. EnergyPlus provides services to input, store, and retrieve curve data and to evaluate curves given values of the independent variables. There are 3 curve objects: CURVE:QUADRATIC, CURVE:CUBIC, and CURVE:BIQUADRATIC.

GetCurveIndex

This function takes a curve name as input and returns an internal pointer to the curve. Curve values will always be accessed via the pointer not the name during the simulation for reasons of efficiency. This function is usually called once for each curve during the input phase.

```
DXCoil(DXCoilNum)%CCapFTemp = GetCurveIndex(Alphas(5))
  IF (DXCoil(DXCoilNum)%CCapFTemp .EQ. 0) THEN
    CALL ShowSevereError('COIL:DX:BF-Empirical not found=' &
                         //TRIM(Alphas(5)))
    ErrorsFound = .TRUE.
  END IF
```

CurveValue

This function takes the curves index and one or two independent variables as input and returns the curve value.

```
!  Get total capacity modifying factor (function of temperature)
!  for off-rated conditions
50 TotCapTempModFac = CurveValue(DXCoil(DXCoilNum)%CCapFTemp,
                                 InletAirWetbulbC, &
                                 OutDryBulbTemp)
```

Fluid Property Services

Fluid properties, rather than a set of embedded calculation, have been implemented with a view to flexibility for EnergyPlus developers and users. Module developers can easily call a set of routines within EnergyPlus and need only tell the users of appropriate inputs for their models. Extensive Reference Data Sets (RDS) of fluid properties are provided for the users.

Using Fluid Property Routines in EnergyPlus Modules

The routines consist of a single module:

FindProperties.f90

You can use the routines anywhere inside EnergyPlus. Just be sure that the module or routine you are in has the following USE statement:

USE FluidProperties

There are eight possible calls to this module. They are listed and explained below. Note that all are real functions (not subroutines) so a call is not necessary—just an assignment statement.

REAL FUNCTION GetSatEnthalpyRefrig(Refrigerant, Temperature, Quality)
REAL FUNCTION GetSatDensityRefrig(Refrigerant, Temperature, Quality)
REAL FUNCTION GetSatSpecificHeatRefrig(Refrigerant, Temperature, Quality)
REAL FUNCTION GetSupHeatEnthalpyRefrig(Refrigerant, Temperature, Pressure)
REAL FUNCTION GetSupHeatPressureRefrig(Refrigerant, Temperature, Enthalpy)
REAL FUNCTION GetSupHeatDensityRefrig(Refrigerant, Temperature, Pressure)
REAL FUNCTION GetSpecificHeatGlycol(Refrigerant, Temperature, Concentration)
REAL FUNCTION GetQualityRefrig(Refrigerant, Temperature, Enthalpy)

Units for these variables are Joules per kilogram for enthalpy, degrees Celsius for temperature, Pascals for pressure, kilograms per cubic meter for density, and Joules per kilogram-degree Celsius for specific heat. Quality and concentration are dimensionless fractions. All variables are considered input variables. The refrigerant name is as it is listed in the input file.

There are several occasions where regions are not loaded in, (or refrigerants that aren't listed above are missing completely). In this case, and some others, the program will call ShowFatalError and deliver a message describing why things failed. Depending on how InputProcessor is hooked up, this message should print upon program exit.

The regions that ARE included are as follows (temperatures in Celsius, pressure in MegaPascals):

Table 1. Regions for Fluid Properties

Refrigerant	Sat. Temp range	Super Temp range*	Super Pressure range*
R11	-70 to 198	-56.67 to 287.78	.001 to 1.379
R11(specheat)	-73 to 187		
R12	-100 to 111.8	-63 to 297	.01 to 8.0
R12(specheat)	-103 to 97		
R22	-90 to 96.15	-67.78 to 243.33	.021 to 3.31
R22(specheat)	-103 to 87		
NH3	-77.67 to 132.3	-45.6 to 198.9	.0345 to 1.93
NH3(specheat)	-73 to 127		
Steam	-60 to 200	100 to 1300	.01 to 60.0
Steam(specheat)	0 to 100		

*Obviously data for all temperatures at all pressures isn't loaded. The entire range of pressures given above will work, but the temperature range for a given pressure will be some subset of the Super Temp range shown above.

Subcooled region actually only returns h(f) or the saturated liquid value at the temperature you input.

Note also that only specific heat is available for glycol solutions. For all other refrigerants, enthalpy, density, and specific heat are available in the saturated region while only enthalpy and density are available in the superheated region.

Fluid Property Data and Expanding the Refrigerants Available to EnergyPlus

The Fluid Property routines have been reengineered to allow other users to add refrigerants to the input file without having to make any changes to the program code. The only requirement on input is that in order to add a new refrigerant, a user must enter a full set of data. The exact definition of a full set of data is given below.

As with all EnergyPlus input, the fluid properties data has both an input data description and a reference data set that must show up in the input file. All of the "standard" refrigerants list above must show up in the in.idf file for it to be available to the rest of the simulation. Below is the description of the input data description syntax for the fluid properties entries.

The first syntax item lists all of the fluids present in an input file and categorizes them as either a refrigerant (such as R11, R12, etc.) or a glycol (such as ethylene glycol, propylene glycol, etc.). A refrigerant or glycol must show up in this list in order to used as a valid fluid in other loops in the input file.

```
FluidNames,
       \unique-object
       \memo list of potential fluid names/types in the input file, maximum of ten
  A1,  \field fluid name 1
       \type alpha
  A2,  \field type of fluid for fluid name 1
       \type choice
       \key REFRIGERANT
       \key GLYCOL
  A3,  \field fluid name 2
       \type alpha
  A4,  \field type of fluid for fluid name 2
       \type choice
       \key REFRIGERANT
       \key GLYCOL
  . . . same thing repeated over and over again . . .
  A19, \field fluid name 10
       \type alpha
  A20; \field type of fluid for fluid name 10
       \type choice
       \key REFRIGERANT
       \key GLYCOL
```

An example of this statement in an input data file is:

```
FluidNames,
  R11, REFRIGERANT,
  R12, REFRIGERANT,
  R22, REFRIGERANT,
  NH3, REFRIGERANT,
  Steam, REFRIGERANT,
  EthyleneGlycol, GLYCOL,
  PropyleneGlycol, GLYCOL;
```

All fluid properties vary with temperature. As a result, the following syntax allows the user to list the temperatures at which the data points are valid. Since in many cases, the temperatures will be similar, this provides a more compact input structure and avoids listing the temperatures multiple times. The name associated with the temperature list is the piece of information that will allow the actual fluid property data statements to refer back to or link to the temperatures. Up to 250 points may be entered with this syntax and temperatures must be entered in ascending order. Units for the temperatures are degrees Celsius. The same temperature list may be used by more than one refrigerant.

```
FluidPropertyTemperatures,
      \memo property values for fluid properties
      \memo list of up to 250 temperatures, note that number of property values must match the
number of properties
      \memo in other words, there MUST be a one-to-one correspondance between the property values
in this list and
      \memo the actual properties list in other syntax
      \units degrees C (for all temperature inputs)
  A1, \field temperature list name
      \type alpha
  N1, \field temperature 1
      \type real
  N2, \field temperature 2
      \type real
  . . . same thing repeated over and over again . . .
  N250; \field temperature 250
        \type real
```

An example of this statement in an input data file is:

```
FluidPropertyTemperatures,
      R11Temperatures,
      -70,-65,-60,-55,-50,-45,-40,-35,-30,-25,-20,-15,-10,-5,0,2,4,6,8,10,12,14,16,18,
      20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,55,60,65,70,75,80,85,90,95,100,
      105,110,115,120,125,130,135,140,145,150,155,160,165,170,175,180,185,190,198;
```

Property data for the saturated region is entered with the following syntax. Before the actual data is entered, this line of input must identify the refrigerant the data is to be associated with, what the data represents (choice of one of three keywords), the phase of the data (either fluid or the difference between fluid and gas), and the temperature list reference that links each data point with a temperature.

```
FluidPropertySaturated,
      \memo fluid properties for the saturated region
  A1, \field fluid name (R11, R12, etc.)
      \reference FluidNames
  A2, \field fluid property type
      \type choice
      \key ENTHALPY     ! Units are J/kg
      \key DENSITY      ! Units are J/kg-K
      \key SPECIFICHEAT ! Units are kg/m^3
  A3, \field fluid phase
      \type choice
      \key FLUID        ! saturated fluid
      \key FLUIDGAS     ! difference between saturated fluid and saturated vapor
  A4, \field temperatures list name
      \reference FluidPropertyTemperatures
  N1, \field property value 1
      \type real
  N2, \field property value 2
      \type real
  . . . same thing repeated over and over again . . .
  N250; \field property value 250
        \type real
```

An example of this statement in an input data file is:

```
FluidPropertySaturated,
      R11,ENTHALPY,FLUID,R11Temperatures, ! Enthalpy in J/kg
      153580,154600,156310,158580,161300,164380,167740,171330,175100,179020,183060,
      187190,191400,195680,200000,201740,203490,205240,207000,208770,210530,212310,
      214080,215870,217650,219860,221230,223030,224830,226630,228860,230250,232060,
      233860,235700,237520,239350,241180,243010,246350,249450,254080,258730,263480,
      268110,272860,277000,282410,287240,292120,297030,302000,307090,312080,317210,
      322400,327670,333020,338460,344010,349680,355500,361480,367690,374100,381060,
      388850,397280,426300;
```

The format of the data for the superheated region is almost identical to that of the saturated region with one addition—a pressure. The pressure is listed before the rest of the data and has units of Pa.

```
FluidPropertySuperheated,
      \memo fluid properties for the saturated region
  A1, \field fluid name (R11, R12, etc.)
      \reference FluidNames
  A2, \field fluid property type
      \type choice
      \key ENTHALPY     ! Units are J/kg
      \key DENSITY      ! Units are J/kg-K
      \key SPECIFICHEAT ! Units are kg/m^3 (Currently this data is not used, entered,
                                            or expected)
  A3, \field temperatures list name
      \reference FluidPropertyTemperatures
  N1, \field pressure
      \memo pressure for this list of properties
      \type real
      \units Pa
      \minimum> 0.0
  N2, \field property value 1
      \type real
  N3, \field property value 2
      \type real
  . . . same thing repeated over and over again . . .
  N251; \field property value 250
        \type real
```

An example of this statement in an input data file is:

```
FluidPropertySuperheated,
    R11,DENSITY,SuperR11Temperatures, ! Density in kg/m^3
    62000., !Pressure = 62000Pa
    0,0,0,0,0,0,0,0.0139,0.0134,0.0129,0.0124,0.012,0.0116,0.0112,0.0109,0.0105,
    0.0102,0.0099,0.0097,0.0094,0.0092,0.0089,0,0,0,0,0,0,0,0,0,0;
```

The format of the data for the glycols is almost identical to that of the superheated region with one exception—concentration replaces pressure. The concentration is listed before the rest of the data and is dimensionless.

```
FluidPropertyConcentration,
      \memo fluid properties for water/other fluid mixtures
  A1, \field fluid name (ethylene glycol, etc.)
      \reference FluidNames
  A2, \field fluid property type
      \type choice
      \key ENTHALPY     ! Units are J/kg   (Currently this data is not used, entered,
                                            or expected)
      \key DENSITY      ! Units are J/kg-K (Currently this data is not used, entered,
                                            or expected)
      \key SPECIFICHEAT ! Units are kg/m^3
  A3, \field temperatures list name
      \reference FluidPropertyTemperatures
  N1, \field concentration
      \memo glycol concentration for this list of properties
      \type real
      \units percentage (as a real decimal)
      \minimum 0.0
      \maximum 1.0
  N2, \field property value 1
      \type real
  N3, \field property value 2
      \type real
  . . . same thing repeated over and over again . . .
  N251; \field property value 250
        \type real
```

An example of this statement in an input data file is:

```
FluidPropertyConcentration,
     PropyleneGlycol,SPECIFICHEAT ,GlycolTemperatures, ! Specific heat in J/kg-K
     0.8, ! Concentration
     2572,2600,2627,2655,2683,2710,2738,2766,2793,2821,2849,2876,2904,2931,2959,
     2987,3014,3042,3070,3097,3125,3153,3180,3208,3236,3263,3291,3319,3346,3374,
     3402,3429,3457;
```

Other Useful Utilities

GetNewUnitNumber

Rather than attempt to keep track of all open files and distribute this list to everyone, we have chosen to use a routine that does this operation. If you need to have a scratch file (perhaps when porting legacy code into EnergyPlus modules), you can use the GetNewUnitNumber function to determine a logical file number for the OPEN and READ/WRITE commands. The function works by looking at all open assigned files and returning a number that isn't being used. This implies that you will OPEN the unit immediately after calling the function (and you should!).

```
Example:
INTEGER, EXTERNAL :: GetNewUnitNumber
…
myunit=GetNewUnitNumber()
OPEN(Unit=myunit,File='myscratch')
```

FindUnitNumber

If you want to find out a unit number for a file you think is already open, you can use the FindUnitNumber function. For example, rather than creating a new unit for debug output, you could latch onto the same unit as currently used for the “eplusout.dbg” file.

```
Example:
INTEGER, EXTERNAL :: FindUnitNumber
…
myunit=FindUnitNumber('eplusout.dbg')
```

If that file is already opened, it will get back the unit number it is currently assigned to. If it is not opened or does not exist, it will go ahead, get a unit number, and OPEN the file. (Should not be used for Direct Access or Binary files!)

Weather Services

All weather data (including Design Day and Location validation) are processed by the WeatherManager module. The SimulationManager invokes the WeatherManager at the proper times to retrieve data. The WeatherManager will retrieve the proper data for the current timestep/hour/day/month from the proper data source (design day definition, weather data file). The WeatherManager puts weather-type data (outside dry bulb, outside wet bulb, humidity, barometric pressure) into the DataEnvironment global data area. There is no need for other modules to call the WeatherManager directly. However, if there is some weather-type data that is needed and not provided in the DataEnvironment global area, contact us.

Error Messages

Three error message routines are provided for the developer, indicating three different levels of error severity: ShowFatalError, ShowSevereError, and ShowWarningError. Each takes a string as an argument. The string is printed out as the message body on the file “eplusout.err”. There are two additional optional arguments, which are file unit numbers on which the message will also be printed. ShowFatalError causes the program to immediately abort.

```
Example:

IF (Construct(ConstrNum)%LayerPoint(Layer) == 0) THEN
   CALL ShowSevereError('Did not find matching material for construct ' &
                         //TRIM(Construct(ConstrNum)%Name)// &
                        ' missing material = ' &
                        //TRIM(ConstructAlphas(Layer)))
   ErrorsFound=.true.
ENDIF
```

Quite a complex message can be constructed using concatenation. These routines can also be used to output numeric fields by writing the numeric variables to a string variable, although this isn't very convenient.

Flags and Parameters

Parameters

Constants that might be useful throughout the program are defined as Fortran parameters in the DataGlobals data module. Examples include *PI*, *PiOvr2*, *DegToRadians*, and *MaxNameLength*. DataHVACGlobals contains parameters that might be useful anywhere in the HVAC simulation. Some examples are *SmallTempDiff* and *SmallMassFlow* that can be used for preventing divide by zero errors. The full set of global parameters can be obtained by examining the modules DataGlobals and DataHVACGlobals.

Simulation Flags

A number of logical flags (variables that are either *true* or *false*) are used throughout EnergyPlus. These flags are normally used to indicate the start or end of a time or simulation period. The following shows a complete list.

In DataGlobals:

BeginSimFlag

Set to true until the actual simulation has begun, set to false after first heat balance time step.

BeginFullSimFlag

Set to true until a full simulation begins (as opposed to a sizing simulation); set to false after the first heat balance time step of the full simulation.

EndSimFlag

Normally false, but set to true at the end of the simulation (last heat balance time step of last hour of last day of last environment).

WarmupFlag

Set to true during the warmup portion of a simulation; otherwise false.

BeginEnvrnFlag

Set to true at the start of each environment (design day or run period), set to false after first heat balance time step in environment. This flag should be used for beginning of environment initializations in most HVAC components. See the example module for correct usage.

EndEnvrnFlag

Normally false, but set to true at the end of each environment (last heat balance time step of last hour of last day of environment).

BeginDayFlag

Set to true at the start of each day, set to false after first heat balance time step in day.

EndDayFlag

Normally false, but set to true at the end of each day (last heat balance time step of last hour of day).

BeginHourFlag

Set to true at the start of each hour, set to false after first heat balance time step in hour.

EndHourFlag

Normally false, but set to true at the end of each hour (last heat balance time step of hour)

BeginTimeStepFlag

Set to true at the start of each heat balance time step, set to false after first HVAC step in the heat balance time step.

Note: never set to false!

EndTimeStepFlag

Normally false, but set to true at the end of each heat balance time step.

Note: never set to true!

In DataHVACGlobals:

FirstTimeStepSysFlag

Set to true at the start of the first HVAC time step within each heat balance time step, false at the end of the HVAC time step. In other words, this flag is true during the first HVAC time step in a heat balance time step, and is false otherwise.

BeginAirLoopFlag

True for first SimAirLoop call; false thereafter.

Note: not used and not set!

In Subroutine SimHVAC:

FirstHVACIteration

True when HVAC solution technique on first iteration, false otherwise. Passed as a subroutine argument into the HVAC equipment simulation driver routines.

The most commonly used logical flag in the HVAC simulation is FirstHVACIteration that is passed around as an argument among the HVAC simulation subroutines. The HVAC simulation is solved iteratively each HVAC time step. FirstHVACIteration is true for the first iteration in each time step and false for the remaining iterations.

Finally, each developer must define and set a “GetInput” flag to make sure input data is read in only once. In the example module Fans the GetInput flag is GetInputFlag; the new developer can follow this example in using such a flag.

Psychrometric services

EnergyPlus has a full complement of psychrometric functions. All the routines are Fortran functions returning a single precision real value. All arguments and results are in SI units.

rhoairfn(pb,tdb,w)

Returns the density of air in kilograms per cubic meter as a function of barometric pressure [pb] (in Pascals), dry bulb temperature [tdb] (in Celsius), and humidity ratio [w] (kilograms of water per kilogram of dry air).

cpairfn(w,T)

Returns the specific heat of air in Joules per kilogram degree Celsius as a function of humidity ratio [w] (kilograms of water per kilogram of dry air) and dry bulb temperature [T] (Celsius).

PSYDPT(TDB,TWB,PB)

Returns the dew point temperature in Celsius as a function of dry bulb temperature [TDB] (Celsius), wet bulb temperature [TWB] (Celsius), and atmospheric pressure [PB] (Pascals).

PSYDPW(W,PB)

Returns the dew point temperature in Celsius as a function of humidity ratio [W] (kilograms of water per kilogram of dry air) and atmospheric pressure [PB] (Pascals).

PSYHTW(TDB,W)

Returns the specific enthalpy of air in Joules per kilogram as a function of dry bulb temperature [TDB] (Celsius) and humidity ratio [W] (kilograms of water per kilogram of dry air).

PSYHTR(TDB,RH,PB)

Returns the specific enthalpy of air in Joules per kilogram as a function of dry bulb temperature [TDB] (Celsius), relative humidity [RH] (fraction), and atmospheric pressure [PB] (Pascals).

PSYTHW(H,W)

Returns the air temperature in Celsius as a function of air specific enthalpy [H] (Joules per kilogram) and humidity ratio [W] (kilograms of water per kilogram of dry air).

PSYRHT(TDB,W,PB)

Returns the relative humifity (fraction) as a function of of dry bulb temperature [TDB] (Celsius), humidity ratio [W] (kilograms of water per kilogram of dry air) and atmospheric pressure [PB] (Pascals).

PSYTWD(TDB,W,PB)

Returns the air wet bulb temperatute in Celsius as a function of dry bulb temperature [TDB] (Celsius), humidity ratio [W] (kilograms of water per kilogram of dry air) and atmospheric pressure [PB] (Pascals).

PSYVTW(TDB,W,PB)

Returns the specific volume in cubic meters per kilogram as a function of dry bulb temperature [TDB] (Celsius), humidity ratio [W] (kilograms of water per kilogram of dry air) and atmospheric pressure [PB] (Pascals).

PSYWDP(TDP,PB)

Returns the humidity ratio in kilograms of water per kilogram of dry air as a function of the dew point temperature [TDB] (Celsius) and atmospheric pressure [PB] (Pascals).

PSYWTH(TDB,H)

Returns the humidity ratio in kilograms of water per kilogram of dry air as a function of dry bulb temperature [TDB] (Celsius) and air specific enthalpy [H] (Joules per kilogram).

PSYWTP(TDB,TWB,PB)

Returns the humidity ratio in kilograms of water per kilogram of dry air as a function of dry bulb temperature [TDB] (Celsius), wet bulb temperature [TWB] (Celsius), and atmospheric pressure [PB] (Pascals).

PSYWTR(TDB,RH,PB)

Returns the humidity ratio in kilograms of water per kilogram of dry air as a function of dry bulb temperature [TDB] (Celsius), relative humidity [RH] (fraction), and atmospheric pressure [PB] (Pascals).

SATUPT(T)

Returns the saturation pressure in Pascals as a function of the air saturation temperature [T] (Celsius).

SATUTH(H,PB)

Returns the air saturation temperature in Celsius as a function of air specific enthalpy [H] (Joules per kilogram) and atmospheric pressure [PB] (Pascals).

SATUTP(P)

Returns the air saturation temperature in Celsius as a function of saturation pressure [P] (Pascals).

HVAC Network

Branches, Connectors, and Nodes

In EnergyPlus, the HVAC system and plant form a network (technically, a graph). The individual pieces of equipment – the fans, coils, chillers, etc. – are connected together by air ducts and fluid pipes. In EnergyPlus nomenclature, the air and fluid circuits are called loops. Specifying how an individual system and plant are connected is done in the EnergyPlus input (IDF) file. The overall structure of the network is defined with Branch and Connector objects. The detail is filled with components and their inlet and outlet nodes. A Branch consists of one or more components arranged sequentially along a pipe or duct. A Connector specifies how three or more branches are connected through a Splitter or Mixer. Nodes connect components along a branch: the outlet node of one component is the inlet node of the next downstream component. The nodes represent conditions at a point on a loop. Each component has one or more inlet and outlet nodes, depending on how many loops it interacts with. A fan, for instance, has one inlet node and one outlet node, since it interacts with a single air loop. A water coil will have 2 inlet and 2 outlet nodes, since it interacts with an air and a fluid loop. Figure 1 shows a diagram of an EnergyPlus HVAC input.

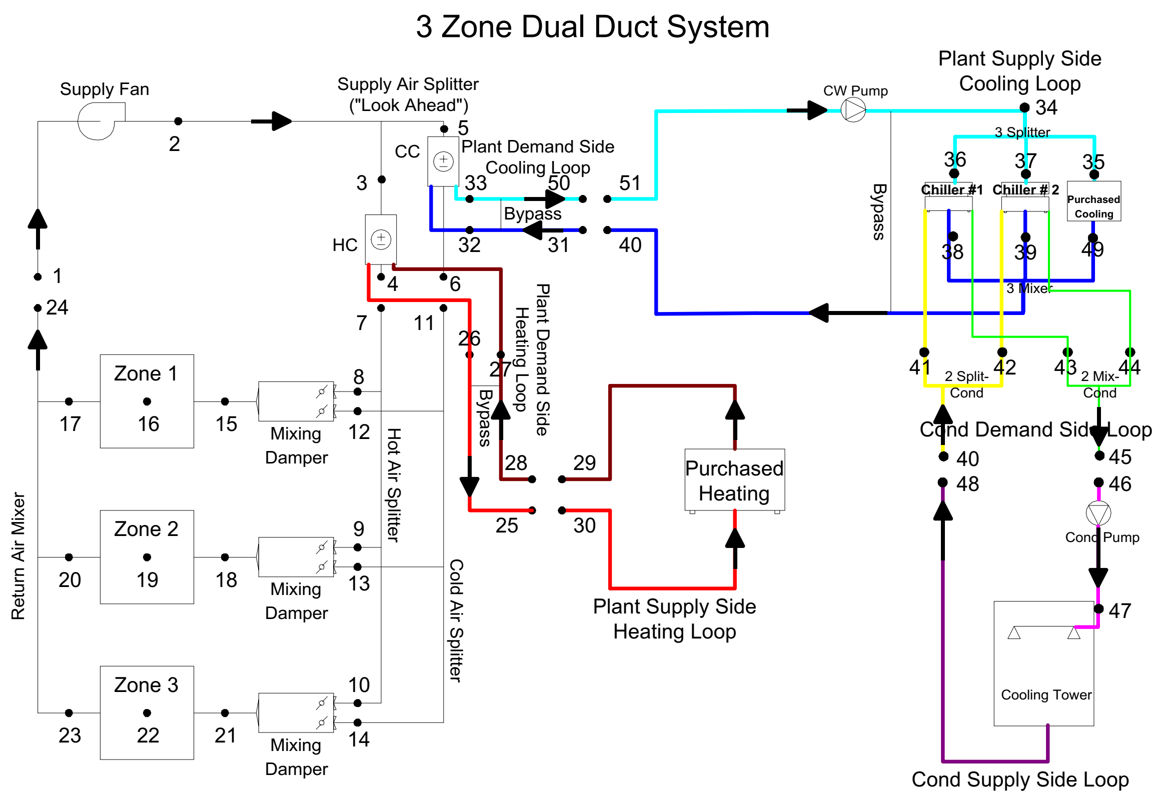

Figure 1. HVAC Input Diagram

As an illustration of how such a network is built up on the IDF, here is the section of the IDF that describes the supply fan, splitter, and heating and cooling coil section of the dual duct air system.

```
BRANCH LIST,
    Dual Duct Air Loop Branches , ! Branch List Name
    Air Loop Main Branch ,        ! Branch Name 1
    Heating Coil Air Sys Branch ,         ! Branch Name 2
    Cooling Coil Air Sys Branch ;         ! Branch Name 3

CONNECTOR LIST,
    Dual Duct Connectors , ! Connector List Name
    SPLITTER ,             ! Type of Connector 1
    DualDuctAirSplitter ;  ! Name of Connector 1

BRANCH,
    Air Loop Main Branch ,  ! Branch Name
    1.3 ,                   ! Maximum Branch Flow Rate
    FAN:SIMPLE:ConstVolume, Supply Fan 1,
      Supply Fan Inlet Node, Supply Fan Outlet Node, PASSIVE ;

BRANCH,
    Heating Coil Air Sys Branch ,  ! Branch Name
    1.3 ,                          ! Maximum Branch Flow Rate
    COIL:Water:SimpleHeating, Main Heating Coil,
      Heating Coil Inlet Node, Heating Coil Outlet Node, ACTIVE ;

BRANCH,
    Cooling Coil Air Sys Branch ,  ! Branch Name
    1.3 ,                          ! Maximum Branch Flow Rate
    COIL:Water:DetailedFlatCooling, Detailed Cooling Coil,
      Cooling Coil Inlet Node, Cooling Coil Outlet Node, ACTIVE ;

SPLITTER,
    DualDuctAirSplitter,
    Air Loop Main Branch,
    Heating Coil Air Sys Branch,
    Cooling Coil Air Sys Branch;

FAN:SIMPLE:ConstVolume,
      Supply Fan 1,  !Fan Name
      FanAndCoilAvailSched, !Fan Schedule
      0.7,           !Fan Efficiency
      600.0,         !Delta Pressure [N/M^2]
      1.3,           !Max Vol Flow Rate  [m^3/Sec]
      0.9,           !motor efficiency
      1.0,           !motor in air stream fraction
      Supply Fan Inlet Node,  ! Inlet Node
      Supply Fan Outlet Node; ! Outlet Node

COIL:Water:DetailedFlatCooling,
      Detailed Cooling Coil, !Name of cooling coil
      CoolingCoilAvailSched, !Cooling Coil Schedule
      1.1,             !Max Water Flow Rate of Coil kg/sec
      6.23816,         !Tube Outside Surf Area
      6.20007018,      !Tube Inside Surf Area
      101.7158224,     !Fin Surf Area
      0.300606367,     !Min Air Flow Area
      0.165097968,     !Coil Depth
      0.43507152,      !Coil Height
      0.001499982,     !Fin Thickness
      0.014449958,     !Tube Inside Diameter
      0.015879775,     !Tube Outside Diameter
      0.385764854,     !Tube Thermal Conductivity
      0.203882537,     !Fin Thermal Conductivity
      0.001814292,     !Fin Spacing
      0.02589977,      !Tube Depth
      6,               !Number of Tube Rows
      16,              !Number of Tubes per Row
      Cooling Coil Water Inlet Node,  ! Coil Water Side Inlet
      Cooling Coil Water Outlet Node, ! Coil Water Side Outlet
      Cooling Coil Inlet Node, !  Coil Air Side Inlet
      Cooling Coil Outlet Node; ! Coil Air Side Outlet

COIL:Water:SimpleHeating,
         Main Heating Coil,     !Name of heating coil
         FanAndCoilAvailSched,  !heating Coil Schedule
         1200.0,                !UA of the Coil
         4.3,                   !Max Water Flow Rate of Coil kg/sec
         Heating Coil Water Inlet, ! Coil Water Side Inlet
         Heating Coil Water Outlet,! Coil Water Side Outlet
         Heating Coil Inlet Node, !  Coil Air Side Inlet
         Heating Coil Outlet Node; ! Coil Air Side Outlet
```

Obviously, the creation of such a system/plant network description is best handled by a GUI. However, for testing purposes a developer may have to create the input for a component by hand and insert it into an existing IDF. Then the developer must be careful to choose unique names for the branches and nodes and make sure the entire network makes physical sense.

Nodes in the simulation

In the EnergyPlus data structure, the nodes are where each component model gets its input and where it places its output. The module *DataLoopNode* contains all the node related data. In particular, the array *Node* contains the state variables and mass flows for all the nodes in the problem being simulated.

```
TYPE NodeData
       CHARACTER(len=MaxNameLength) :: FluidType
       REAL     :: Temp
       REAL     :: TempMin
       REAL     :: TempMax
       REAL     :: TempSetPoint
       REAL     :: MassFlowRate
       REAL     :: MassFlowRateMin
       REAL     :: MassFlowRateMax
       REAL     :: MassFlowRateMinAvail
       REAL     :: MassFlowRateMaxAvail
       REAL     :: MassFlowRateSetPoint
       REAL     :: Quality
       REAL     :: Press
       REAL     :: Enthalpy
       REAL     :: HumRat
       REAL     :: HumRatMin
       REAL     :: HumRatMax
       REAL     :: HumRatSetPoint
 END TYPE NodeData

 !dim to num nodes in SimHVAC
```

```
  TYPE (NodeData), ALLOCATABLE, DIMENSION(:) :: Node
```

In our example module *NewHVACComponent,* the subroutine *InitNewHVACComponent* is responsible for obtaining the input data from the inlet node(s) and putting it into the component data structure for use in C*alcNewHVACComponent*. Then *UpdateNewHVACComponent* takes the calculated data and moves it to the outlet nodes for use by other components. EnergyPlus component models are assumed to be direct models: inlets are input to the calculation and outlets are output from the calculations.

Data Flow in an HVAC Component Module

The data in an EnergyPlus HVAC component module resides in three places.

1. The component inlet nodes – this is where the data input to the model resides.
2. The component internal data structure(s) – one or more arrays of data structures which contain all the data needed for the component simulation. This includes data from the input file, data from the inlet nodes, and any schedule values. In addition, these data structure(s) store the results of the calculation.
3. The component outlet nodes – data is moved from the internal data structure(s) to the outlet nodes at the completion of each component simulation

The data flows from the inlet nodes into the component internal data structure(s) and then into the outlet nodes. Let us see how this works in our example module Fans.

At the start of the module, the component internal data structure is defined.

```
TYPE FanEquipConditions
  CHARACTER(len=MaxNameLength) :: FanName  ! Name of the fan
  CHARACTER(len=MaxNameLength) :: FanType  ! Type of Fan ie. Simple, Vane axial, Centrifugal, etc.
  CHARACTER(len=MaxNameLength) :: Schedule ! Fan Operation Schedule
  CHARACTER(len=MaxNameLength) :: Control  ! ie. Const Vol, Variable Vol
  Integer      :: SchedPtr ! Pointer to the correct schedule
  REAL         :: InletAirMassFlowRate  !MassFlow through the Fan being Simulated [kg/Sec]
  REAL         :: OutletAirMassFlowRate
  Real         :: MaxAirFlowRate  !Max Specified Volume Flow Rate of Fan [m^3/sec]
  Real         :: MinAirFlowRate  !Min Specified Volume Flow Rate of Fan [m^3/sec]
  REAL         :: MaxAirMassFlowRate ! Max flow rate of fan in kg/sec
  REAL         :: MinAirMassFlowRate ! Min flow rate of fan in kg/sec
  REAL         :: InletAirTemp
  REAL         :: OutletAirTemp
  REAL         :: InletAirHumRat
  REAL         :: OutletAirHumRat
  REAL         :: InletAirEnthalpy
  REAL         :: OutletAirEnthalpy
  REAL         :: FanPower             !Power of the Fan being Simulated [kW]
  REAL         :: FanEnergy            !Fan energy in [kJ]
  REAL         :: DeltaTemp            !Temp Rise across the Fan [C]
  REAl         :: DeltaPress           !Delta Pressure Across the Fan [N/M^2]
  REAL         :: FanEff               !Fan total efficiency; motor and mechanical
  REAL         :: MotEff               !Fan motor efficiency
  REAL         :: MotInAirFrac         !Fraction of motor heat entering air stream
  REAL, Dimension(5):: FanCoeff        !Fan Part Load Coefficients to match fan type
  ! Mass Flow Rate Control Variables
  REAL         :: MassFlowRateMaxAvail
  REAL         :: MassFlowRateMinAvail
  INTEGER      :: InletNodeNum
  INTEGER      :: OutletNodeNum
 END TYPE FanEquipConditions

!MODULE VARIABLE DECLARATIONS:
  INTEGER :: NumFans     ! The Number of Fans found in the Input
  TYPE (FanEquipConditions), ALLOCATABLE, DIMENSION(:) :: Fan
```

In this case, there is only one structure that stores all of the fan data. We could have chosen to divide this rather large structure up into separate structures – one for input file data, one for inlet data, and one for outlet data, for instance. Note that in Fortran 90 structures are called defined type. The TYPE – END TYPE construct defines a new data structure. Then an allocatable array *Fan* of the defined type is created. This one-dimensional array will contain an entry for each fan in the problem.

The internal data array is allocated (sized) in the “GetInput” routine GetFanInput.

```
NumSimpFan   = GetNumObjectsFound('FAN:SIMPLE:CONSTVOLUME')
    NumVarVolFan = GetNumObjectsFound('FAN:SIMPLE:VARIABLEVOLUME')
    NumOnOff = GetNumObjectsFound('FAN:SIMPLE:ONOFF')
    NumZoneExhFan = GetNumObjectsFound('ZONE EXHAUST FAN')
    NumFans = NumSimpFan + NumVarVolFan + NumZoneExhFan+NumOnOff
    IF (NumFans.GT.0) ALLOCATE(Fan(NumFans))
```

The remainder of the “GetInput” routine moves input file data into the Fan array. The “Init” routine transfers data from the inlet nodes into the same array in preparation for performing the calculation.

```
  ! Load the node data in this section for the component simulation
  !
  !First need to make sure that the massflowrate is between the max and min avail.
  IF (Fan(FanNum)%FanType .NE. 'ZONE EXHAUST FAN') THEN
    Fan(FanNum)%InletAirMassFlowRate = Min(Node(InletNode)%MassFlowRate, &
                                           Fan(FanNum)%MassFlowRateMaxAvail)
    Fan(FanNum)%InletAirMassFlowRate = Max(Fan(FanNum)%InletAirMassFlowRate, &
                                           Fan(FanNum)%MassFlowRateMinAvail)
  ELSE  ! zone exhaust fans - always run at the max
    Fan(FanNum)%MassFlowRateMaxAvail = Fan(FanNum)%MaxAirMassFlowRate
    Fan(FanNum)%MassFlowRateMinAvail = 0.0
    Fan(FanNum)%InletAirMassFlowRate = Fan(FanNum)%MassFlowRateMaxAvail
  END IF

  !Then set the other conditions
  Fan(FanNum)%InletAirTemp         = Node(InletNode)%Temp
  Fan(FanNum)%InletAirHumRat       = Node(InletNode)%HumRat
  Fan(FanNum)%InletAirEnthalpy     = Node(InletNode)%Enthalpy
```

The “Calc” routines do the actual component simulation. All the data they need has been stored in the internal data array ready to be used. The results of the calculation are, in this case, stored in the same array. The “Calc” routine always does pure calculation/simulation – it never retrieves or stores data.

```
DeltaPress = Fan(FanNum)%DeltaPress
   FanEff     = Fan(FanNum)%FanEff

   ! For a Constant Volume Simple Fan the Max Flow Rate is the Flow Rate for the fan
   Tin        = Fan(FanNum)%InletAirTemp
   Win        = Fan(FanNum)%InletAirHumRat
   RhoAir     = RhoAirFn(OutBaroPress,Tin,Win)
   MassFlow   = MIN(Fan(FanNum)%InletAirMassFlowRate,Fan(FanNum)%MaxAirMassFlowRate)
   MassFlow   = MAX(MassFlow,Fan(FanNum)%MinAirMassFlowRate)
   !
   !Determine the Fan Schedule for the Time step
 If((GetCurrentScheduleValue(Fan(FanNum)%SchedPtr) .gt. 0.0) .and. &
       (Massflow .gt. 0.0)) Then
   !Fan is operating
   Fan(FanNum)%FanPower = MassFlow*DeltaPress/(FanEff*RhoAir) ! total fan power
   FanShaftPower = Fan(FanNum)%MotEff * Fan(FanNum)%FanPower  ! power delivered to shaft
   PowerLossToAir = FanShaftPower + (Fan(FanNum)%FanPower - FanShaftPower) * &
     Fan(FanNum)%MotInAirFrac
   Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy + PowerLossToAir/MassFlow
   ! This fan does not change the moisture or Mass Flow across the component
   Fan(FanNum)%OutletAirHumRat       = Fan(FanNum)%InletAirHumRat
   Fan(FanNum)%OutletAirMassFlowRate = MassFlow
   Fan(FanNum)%OutletAirTemp = Psythw(Fan(FanNum)%OutletAirEnthalpy,Fan(FanNum)%OutletAirHumRat)

 Else
   !Fan is off and not operating no power consumed and mass flow rate.
   Fan(FanNum)%FanPower = 0.0
   FanShaftPower = 0.0
   PowerLossToAir = 0.0
   Fan(FanNum)%OutletAirMassFlowRate = 0.0
   Fan(FanNum)%OutletAirHumRat       = Fan(FanNum)%InletAirHumRat
   Fan(FanNum)%OutletAirEnthalpy     = Fan(FanNum)%InletAirEnthalpy
   Fan(FanNum)%OutletAirTemp = Fan(FanNum)%InletAirTemp
   ! Set the Control Flow variables to 0.0 flow when OFF.
   Fan(FanNum)%MassFlowRateMaxAvail = 0.0
   Fan(FanNum)%MassFlowRateMinAvail = 0.0

  End If
```

Finally, the “Update” routine (UpdateFan) moves the results from the internal data array into the outlet node(s).

```
   OutletNode = Fan(FanNum)%OutletNodeNum
   InletNode = Fan(FanNum)%InletNodeNum

   ! Set the outlet air nodes of the fan
   Node(OutletNode)%MassFlowRate  = Fan(FanNum)%OutletAirMassFlowRate
   Node(OutletNode)%Temp          = Fan(FanNum)%OutletAirTemp
   Node(OutletNode)%HumRat        = Fan(FanNum)%OutletAirHumRat
   Node(OutletNode)%Enthalpy      = Fan(FanNum)%OutletAirEnthalpy
   ! Set the outlet nodes for properties that just pass through & not used
   Node(OutletNode)%Quality         = Node(InletNode)%Quality
   Node(OutletNode)%Press           = Node(InletNode)%Press

   ! Set the Node Flow Control Variables from the Fan Control Variables
   Node(OutletNode)%MassFlowRateMaxAvail = Fan(FanNum)%MassFlowRateMaxAvail
   Node(OutletNode)%MassFlowRateMinAvail = Fan(FanNum)%MassFlowRateMinAvail

   IF (Fan(FanNum)%FanType .EQ. 'ZONE EXHAUST FAN') THEN
     Node(InletNode)%MassFlowRate = Fan(FanNum)%InletAirMassFlowRate
   END IF
```

Certain data items must always be transferred from inlet nodes to outlet nodes even if the data item is unaltered by the component model. The data items that must be transferred are:

1. Temp
2. HumRat
3. Enthalpy
4. Press
5. MassFlowRate
6. MassFlowRateMaxAvail
7. MassFlowRateMinAvail

Node Mass Flow Variables

The node mass flow variables merit a little more discussion. Five mass flow variables are defined at each node. They are: MassFlowRate, MassFlowRateMin, MassFlowRateMax, MassFlowRateMinAvail and MassFlowRateMaxAvail. These variables hold loop mass flow rate information according to the following definitions.

- MassFlowRate – this node variable holds the simulation mass flow rate for the current timestep. The component simulation retrieves this mass flow rate from its inlet node ("Init" routine) and uses it as the initial mass flow rate in the simulation. The component simulation may or may not change the mass flow rate. In any case, it writes the value out to the exit node in the "Update" routine.

- MassFlowRateMax, MassFlowRateMin – These node variables hold the maximum possible and the minimum allowable flow rates for a particular component. As such, they represent the "hardware limit" on the flow rate for the component. By convention, these variables are stored at the component outlet node. Since components share their nodes (the outlet node of one component is the inlet node of the next component), the protocol must be strictly followed. These variables are set by each component at the beginning of the simulation and are never reset thereafter.

- MassFlowRateMaxAvail, MassFlowRateMinAvail – these node variables represent the *loop* maximum and minimum flow rate for the current configuration of the loop on which the component resides. All components should change the max/min available values so that they lie within the max/min flow rate range for the component. The component simulation reads the current loop min/max available flow rate from its inlet node ("Init" routine) and writes the updated min/max available flow rate to its outlet node ("Update" routine).

The component simulation retrieves MassFlowRate, MassFlowRateMinAvail, and MassFlowRateMaxAvail along with other node variables from its input node prior to the simulation of the component. The flow rate must be checked and if necessary adjusted prior to the simulation: the MassFlowRate must be bounded by MassFlowRateMaxAvail and MassFlowRateMinAvail, which in turn must be bounded by MassFlowRateMax and MassFlowRateMin for the component. The following steps should be followed in the initialization stage of the component simulation.

a) Compare the MassFlowRateMinAvail and MassFlowRateMaxAvail retrieved from the input node with the MassFlowRateMin and MassFlowRateMax for your component. If of the retrieved values are out of bounds, replace that value with either the Max or Min for the component.

b) Compare the MassFlowRate retrieved from the inlet node with the MassFlowRateMinAvail and MassFlowRateMaxAvail. If MassFlowRate is not bounded by MassFlowRateMinAvail and MassFlowRateMaxAvail, reset MassFlowRate to the nearest boundary value.

If the component model calculates MassFlowRate, it must be bounded by MassFlowRateMin and MassFlowRateMax. Following the component simulation, the MassFlowRate and the bounding MassFlowRateMinAvail and MassFlowRateMaxAvail should be written to the outlet node along with the other updated loop variables.

Output

There are several output files available in EnergyPlus. As you can see in the Appendix A, DataGlobals contains OutputFileStandard, OutputFileInits, and OutputFileDebug.

OutputFileDebug is initialized very early in the EnergyPlus execution and is available for any debugging output the developer might need.

OutputFileInits is intended for "one-time" outputs and has not been extensively used to date. The structure will be similar to the IDD/IDF structure in that there will be a "definition" line followed by the data being reported.

OutputFileStandard is the usual output file from EnergyPlus. You can read more details from the *Guide for Interface Developers* document. There is an Output section there as well as the keywords for obtaining those outputs.

How Do I Output My Variables?

Module developers are responsible for "setting" up the variables that will appear in the OutputFileStandard.

To do this is very simple. All you need to do is place a simple call to *SetupOutputVariable* for each variable to be available for reporting into your module. This call should be done only once for each Variable/KeyedValue pair (see below). For HVAC and Plant components, this call is usually at the end of the "GetInput" subroutine. See the example module for an illustration of this. Other calls in the simulation routines will invoke the EnergyPlus *OutputProcessor* automatically at the proper time to have the data appear in the OutputFileStandard.

For you the call is:

```
Call SetupOutputVariable(VariableName,ActualVariable,  &
                         IndexTypeKey, VariableTypeKey,KeyedValue,ReportFreq &
                         ResourceTypeKey,EndUseKey,GroupKey)
```

Interface statements allow for the same call to be used for either real or integer "ActualVariable" variables. A few examples from EnergyPlus and then we will define the arguments:

```
CALL SetupOutputVariable('Outdoor Dry Bulb [C]', &
                          OutDryBulbTemp,'Zone', &
                          'Average','Environment')

CALL SetupOutputVariable('Mean Air Temperature[C]', &
                         ZnRpt(Loop)%MeanAirTemp,'Zone', &
                         'State',Zone(Loop)%Name)

CALL SetupOutputVariable('Fan Coil Heating Energy[J]', &
                         FanCoil(FanCoilNum)%HeatEnergy,'System', &
                         'Sum',FanCoil(FanCoilNum)%Name)
CALL SetupOutputVariable('Humidifier Electric Consumption[J]',
                         Humidifier(HumNum)%ElecUseEnergy, &
                         'System','Sum', &
                         Humidifier(HumNum)%Name,&
                         ResourceTypeKey='ELECTRICITY',&
                         EndUseKey = 'HUMIDIFIER',&
                         GroupKey = 'System')
```

SetupOutputVariable Arguments	Description
VariableName	String name of variable, units should be included in []. If no units, use []
ActualVariable	This should be the actual variable that will store the value. The OutputProcessor sets up a pointer to this variable, so it will need to be a SAVEd variable if in a local routine. As noted in examples, can be a simple variable or part of an array/derived type.
IndexTypeKey	When this variable has its proper value. ‘Zone’ is used for variables that will have value on the global timestep (alias “HeatBalance”). ‘HVAC’ is used for variables that will have values calculated on the variable system timesteps (alias “System”, “Plant”)
VariableTypeKey	Two kinds of variables are produced. ‘State’ or ‘Average’ are values that are instantaneous at the timestep (zone air temperature, outdoor weather conditions). ‘NonState’ or ‘Sum’ are values which need to be summed for a period (energy).
KeyedValue	Every variable to be reported needs to have an associated keyed value. Zone Air Temperature is available for each Zone, thus the keyed value is the Zone Name.
ReportFreq	This optional argument should only be used during debugging of your module but it is provided for the developers so that these variables would always show up in the OutputFile. (All other variables must be requested by the user).
ResourceTypeKey	Meter Resource Type; an optional argument used for including the variable in a meter. The meter resource type can be 'Electricity', ‘Gas’, ‘Coal’, ‘FuelOil#1’, ‘FuelOil#2’, ‘Propane’, ‘Water’, or ‘EnergyTransfer’.
EndUseKey	Meter End Use Key; an optional argument used when the variable is included in a meter. The end us keys can be: 'GeneralLights', ‘TaskLights’, 'ExteriorLights', 'Heating', ‘Cooling’, 'DHW', 'Cogeneration', 'ExteriorEquipment', 'ZoneSource', 'PurchasedHotWater', 'PurchasedChilledWater', 'Fans', 'HeatingCoils', 'CoolingCoils', 'Pumps', 'Chillers', 'Boilers', 'Baseboard', 'HeatRejection', 'Humidifier', 'HeatRecovery', or 'Miscellaneous'.
GroupKey	Meter Super Group Key; an optional arument used when the variable is included in a meter. The group key denotes whether the variable belongs to the building, system, or plant.The choices are: 'Building', 'HVAC' or 'Plant'.

Table 2. SetupOutputVariable Arguments

As described in the *Input Output Reference*, not all variables may be available in any particular simulation. Only those variables that will have values generated will be available for reporting. In the IDF, you can include a “Report Variable Dictionary” command that will produce the eplusout.rdd file containing all the variables with their IndexTypeKeys. This list can be used to tailor the requests for values in the OutputFileStandard.

Output Variable Dos and Don’ts

For general output variables there aren’t many rules. For meter output variables there are quite a few. Here are some tips to keep you out of trouble.

What Variables Should I Output?

The choice of variables to output is really up to the developer. Since variables don’t appear on the output file unless requested by the user in the IDF input file, it is better to “SetUp” too many rather than too few. For an HVAC component one should generally output the heating and cooling outputs of the component both in terms of energy and power. Energy is always output in Joules, power in Watts. If there is humidification or dehumidification both total and sensible cooling should be reported. Any electricity or fuel consumed by a component should be reported out, again both in terms of energy (Joules) and power (Watts). For HVAC components in most cases reporting inlet and outlet temperatures and humidites is unnecessary since these quantities can be obtained from the system node outputs.

Output Variable Naming Conventions

We have tried to obtain some consistency in variable names by defining some naming conventions. The heating and/or cooling output is always reported as:

```
<component-type> Heating Rate[W]
<component-type> Heating Energy[J]
<component-type> Total Cooling Rate[W]
<component-type> Total Cooling Energy[W]
<component-type> Sensible Cooling Rate[W]
<component-type> Sensible Cooling Energy[J]
```

Fuel and electricity consumption is reported as:

```
<component-type> Electric Power[W]
<component-type> Electric Consumption[J]
<component-type> Gas Consumption Rate[W]
<component-type> Gas Consumption[J]
```

Water addition is reported as:

```
<component-type> Water Consumption Rate[m3/s]
<component-type> Water Consumption[m3]
```

Units are always strictly SI and no abbreviations are allowed in the variable name. <component-type> is the type of component. It should not be the actual object class name from the IDD file, but rather one step of generality above this. For example for fancoils we have:

```
Fan Coil Total Cooling Energy[J]
```

Here <component-type> is “Fan Coil”, not FAN COIL UNIT:4 PIPE.

What are Meters?

In EnergyPlus meters are an additional output reporting capability. A meter is a way of grouping similar output variables. Meters are output variables just like ordinary output variables except that they sum or average a collection of ordinary output variables. In EnergyPlus the meter variables serve two purposes.

1. Providing output of fuel and electricity consumption by end use categories and at the system plant, building and facility level.
2. Providing a way of summing heating or cooling outputs for a category of components. The resource type EnergyTransfer is used for this purpose. An example would be reporting out the sum of the heating energy from all the heating coils in a system.

How Do I Create A Meter?

Meter output variables are created at the same time and in the same manner a ordinary output variables. SetupOutputVariable is called but the optional arguments ResourceTypeKey, EndUseKey, and GroupKey must be used in addition to the usual arguments. For example, in the electric steam humidifier module

```
CALL SetupOutputVariable('Humidifier Electric Consumption[J]', &
  Humidifier(HumNum)%ElecUseEnergy, 'System','Sum', &
  Humidifier(HumNum)%Name)
```

creates an output variable labeled 'Humidifier Electric Consumption[J]' with the value of Humidifier(HumNum)%ElecUseEnergy.

```
CALL SetupOutputVariable('Humidifier Electric Consumption[J]', &
  Humidifier(HumNum)%ElecUseEnergy, 'System','Sum', &
  Humidifier(HumNum)%Name, &
  ResourceTypeKey='ELECTRICITY',EndUseKey = 'HUMIDIFIER', &
  GroupKey = 'System')
```

Creates the same output variable but in addition creates a meter output variable Humidifier:Electricity [J]. This variable will contain the sum of all the electricity consumption of the humidifiers in the system. In addition, this electrical consumption will be added into the meter variables Electricity:HVAC [J] and Electricity:Facility [J].

Rules for Meter Variables

There are a number of rules developers must follow in order to account for all electricity and fuel consumption as well as to prevent consumables being double counted.

1. Electricity and fuel meters must always be defined at the simple component level. Some EnergyPlus components are compound components: they are built up from simple components. Examples are fan coils (composed of heating coils, cooling coils, and fans), terminal units etc. Some example simple components are heating and cooling coils, fans, humidifiers etc. Electricity and fuel consumption should always be metered at the simple component level and never at the compound component level. This prevents double counting of the fuel or energy consumption.
2. A variable should be metered once only. This means a variable can be assigned to only one resource type and to only one end use category.
3. Energy Transfer should be metered in the same way as fuel or electricity use. Energy Transfer meters should only be defined for simple components and should be assigned the same end use category as the fuel or electricity consumption.
4. All fuel and electricity consumption must be put in some (one) meter.

5. Use Energy Transfer judiciously; if in doubt, leave it out.

Important Rules for Module Developers

1. INITIALIZE!!!!! INITIALIZE either fully or "invalidly" when you ALLOCATE the array/derived type. Two items have been set up to help you: BigNumber and DBigNumber are in DataGlobals. They get initialized before anything happens in the main routine (EnergyPlus). An invalid initialization will use one of these, appropriately, -- don't know yet if this will trigger an exception when used...

2. Warning errors during "get input" probably aren't. Each GetInput routine should be structured so that errors detected (such as an invalid schedule name which currently is just a warning) cause a fatal error after all the input for that item/module/etc is gotten. (See HBManager, BaseboardRadiator, others) In addition, don't make GetInputFlag a module variable. Make it as "local" as possible. Look at BaseboardRadiator for an example.

3. Error messages during simulation should be semi-intelligent. No one wants to see 5,000 messages saying "this flow invalid". If it might happen a lot (especially during debugging), count them and only put out a message every 50 or so. (See some routine in Resolver code). Also, if you are putting the same message in two modules, identify the error message with some designation. For Example, CALL ShowWarningError('SimRoutinename: this condition happened again') will help everyone track it down.

4. Use the templates for documentation! Modules, subroutines, functions templates all have been checked into StarTeam. Use them. Put INTENTs on your Subroutine Arguments. Document variables.

Appendix A. DataGlobals Module

```
MODULE DataGlobals      ! EnergyPlus Data-Only Module

        ! MODULE INFORMATION:
        !       AUTHOR         Rick Strand
        !       DATE WRITTEN   January 1997
        !       MODIFIED       May 1997 (RKS) Added Weather Variables
        !       MODIFIED       December 1997 (RKS,DF,LKL) Split into DataGlobals and
        !                      DataEnvironment
        !       MODIFIED       February 1999 (FW) Added NextHour, WGTNEXT, WGTNOW
        !       MODIFIED       September 1999 (LKL) Rename WGTNEXT,WGTNOW for clarity
        !       RE-ENGINEERED  na

        ! PURPOSE OF THIS MODULE:
        ! This data-only module is a repository for all variables which are considered
        ! to be "global" in nature in EnergyPlus.

        ! METHODOLOGY EMPLOYED:
        ! na

        ! REFERENCES:
        ! na

        ! OTHER NOTES:
        ! na

        ! USE STATEMENTS:
        ! None!--This module is USEd by all other modules; it should not USE anything.

IMPLICIT NONE   ! Enforce explicit typing of all variables

PUBLIC          ! By definition, all variables which are placed in this data
                ! -only module should be available to other modules and routines.
                ! Thus, all variables in this module must be PUBLIC.


          ! MODULE PARAMETER DEFINITIONS:
          !Loop fluid identifiers
!INTEGER, PARAMETER:: Water     =1  ! saturated to compressed liquid
!INTEGER, PARAMETER:: Steam     =2  ! saturated and superheated conditions
!INTEGER, PARAMETER:: EthGlycol =3  ! ethylene glycol brine of various concentrations
!INTEGER, PARAMETER:: Propglycol=4  ! propylene glycol brine of various concentrations
!INTEGER, PARAMETER:: CaChloride=5  ! calcium chloride brine of various concentrations
!INTEGER, PARAMETER:: NaChloride=6  ! sodium chloride brine of various concentrations
!INTEGER, PARAMETER:: Refrig    =7  ! currently, "generic" refrigerant...the parameter
                                   ! list may be extended to include a number of specific
                                   ! refrigerants

INTEGER, PARAMETER :: BeginDay = 1
INTEGER, PARAMETER :: DuringDay = 2
INTEGER, PARAMETER :: EndDay = 3
INTEGER, PARAMETER :: EndZoneSizingCalc = 4
INTEGER, PARAMETER :: EndSysSizingCalc = 5

INTEGER, PARAMETER :: ZoneTSReporting=1  ! value for Zone Time Step Reporting-UpdateDataAndReport
INTEGER, PARAMETER :: HVACTSReporting=2  ! value for HVAC Time Step Reporting-UpdateDataAndReport

REAL, PARAMETER    :: PI= 3.141592653589793   ! Pi
REAL, PARAMETER    :: PiOvr2 = PI/2.          ! Pi/2
REAL, PARAMETER    :: DegToRadians = PI/180.  ! Conversion for Degrees to Radians
INTEGER, PARAMETER :: MaxNameLength = 60 ! Maximum Name Length in Characters -- should be the same
                                          ! as MaxAlphaArgLength in InputProcessor module

REAL, PARAMETER    :: InitConvTemp = 5.05     ! [deg C], standard init vol to mass flow conversion
!                                              temp
INTEGER, PARAMETER :: MaxSlatAngs = 19
```

```
          ! DERIVED TYPE DEFINITIONS
          ! na


          ! INTERFACE BLOCK SPECIFICATIONS
  INTERFACE
    SUBROUTINE ShowMessage(Message,Unit1,Unit2)
    !  Use when you want to create your own message for the error file.
    CHARACTER(len=*) Message    ! Message automatically written to "error file"
    INTEGER, OPTIONAL :: Unit1  ! Unit number of open formatted file for message
    INTEGER, OPTIONAL :: Unit2  ! Unit number of open formatted file for message
    END SUBROUTINE
  END INTERFACE
  INTERFACE
    SUBROUTINE ShowContinueError(Message,Unit1,Unit2)
    !  Use when you are "continuing" an error message over several lines.
    CHARACTER(len=*) Message    ! Message automatically written to "error file"
    INTEGER, OPTIONAL :: Unit1  ! Unit number of open formatted file for message
    INTEGER, OPTIONAL :: Unit2  ! Unit number of open formatted file for message
    END SUBROUTINE
  END INTERFACE
  INTERFACE
    SUBROUTINE ShowFatalError(Message,Unit1,Unit2)
    !  Use when you want the program to terminate after writing messages
    !  to appropriate files
    CHARACTER(len=*) Message    ! Message automatically written to "error file"
    INTEGER, OPTIONAL :: Unit1  ! Unit number of open formatted file for message
    INTEGER, OPTIONAL :: Unit2  ! Unit number of open formatted file for message
    END SUBROUTINE
  END INTERFACE
  INTERFACE
    SUBROUTINE ShowSevereError(Message,Unit1,Unit2)
    !  Use for "severe" error messages.  Might have several severe tests and then terminate.
    CHARACTER(len=*) Message    ! Message automatically written to "error file"
    INTEGER, OPTIONAL :: Unit1  ! Unit number of open formatted file for message
    INTEGER, OPTIONAL :: Unit2  ! Unit number of open formatted file for message
    END SUBROUTINE
  END INTERFACE
  INTERFACE
    SUBROUTINE ShowWarningError(Message,Unit1,Unit2)
    !  Use for "warning" error messages.
    CHARACTER(len=*) Message    ! Message automatically written to "error file"
    INTEGER, OPTIONAL :: Unit1  ! Unit number of open formatted file for message
    INTEGER, OPTIONAL :: Unit2  ! Unit number of open formatted file for message
    END SUBROUTINE
  END INTERFACE

  INTERFACE SetupOutputVariable
    SUBROUTINE
SetupRealOutputVariable(VariableName,ActualVariable,IndexTypeKey,VariableTypeKey,KeyedValue,   &
                                       ReportFreq,ResourceTypeKey,EndUseKey,GroupKey)
      CHARACTER(len=*), INTENT(IN) :: VariableName   ! String Name of variable
      REAL, INTENT(IN), TARGET     :: ActualVariable ! Actual Variable, used to set up pointer
      CHARACTER(len=*), INTENT(IN) :: IndexTypeKey    ! Zone, HeatBalance=1, HVAC, System, Plant=2
      CHARACTER(len=*), INTENT(IN) :: VariableTypeKey ! State, Average=1, NonState, Sum=2
      CHARACTER(len=*), INTENT(IN) :: KeyedValue     ! Associated Key for this variable
      CHARACTER(len=*), INTENT(IN), OPTIONAL :: ReportFreq      ! Internal use -- causes reporting
at this freqency
      CHARACTER(len=*), INTENT(IN), OPTIONAL :: ResourceTypeKey ! Meter Resource Type
(Electricity, Gas, etc)
      CHARACTER(len=*), INTENT(IN), OPTIONAL :: EndUseKey       ! Meter End Use Key (Task Lights,
Heating, Cooling, etc)
      CHARACTER(len=*), INTENT(IN), OPTIONAL :: GroupKey        ! Meter Super Group Key (Building,
System, Plant)
    END SUBROUTINE
    SUBROUTINE
SetupIntegerOutputVariable(VariableName,IntActualVariable,IndexTypeKey,VariableTypeKey,KeyedValue,
ReportFreq)
      CHARACTER(len=*), INTENT(IN) :: VariableName   ! String Name of variable
      INTEGER, INTENT(IN), TARGET  :: IntActualVariable ! Actual Variable, used to set up pointer
      CHARACTER(len=*), INTENT(IN) :: IndexTypeKey    ! Zone, HeatBalance=1, HVAC, System, Plant=2
      CHARACTER(len=*), INTENT(IN) :: VariableTypeKey ! State, Average=1, NonState, Sum=2
      CHARACTER(len=*), INTENT(IN) :: KeyedValue     ! Associated Key for this variable
      CHARACTER(len=*), INTENT(IN), OPTIONAL :: ReportFreq     ! Internal use -- causes reporting
at this freqency
    END SUBROUTINE
    SUBROUTINE
SetupRealOutputVariable_IntKey(VariableName,ActualVariable,IndexTypeKey,VariableTypeKey,KeyedValue
,  &
                                              ReportFreq,ResourceTypeKey,EndUseKey,GroupKey)
      CHARACTER(len=*), INTENT(IN) :: VariableName   ! String Name of variable
      REAL, INTENT(IN), TARGET     :: ActualVariable ! Actual Variable, used to set up pointer
      CHARACTER(len=*), INTENT(IN) :: IndexTypeKey    ! Zone, HeatBalance=1, HVAC, System, Plant=2
      CHARACTER(len=*), INTENT(IN) :: VariableTypeKey ! State, Average=1, NonState, Sum=2
      INTEGER, INTENT(IN)          :: KeyedValue     ! Associated Key for this variable
      CHARACTER(len=*), INTENT(IN), OPTIONAL :: ReportFreq      ! Internal use -- causes reporting
at this freqency
      CHARACTER(len=*), INTENT(IN), OPTIONAL :: ResourceTypeKey ! Meter Resource Type
(Electricity, Gas, etc)
      CHARACTER(len=*), INTENT(IN), OPTIONAL :: EndUseKey       ! Meter End Use Key (Task Lights,
Heating, Cooling, etc)
      CHARACTER(len=*), INTENT(IN), OPTIONAL :: GroupKey        ! Meter Super Group Key (Building,
System, Plant)
    END SUBROUTINE
  END INTERFACE

  INTERFACE SetupRealInternalOutputVariable
    INTEGER FUNCTION
SetupRealInternalOutputVariable(VariableName,ActualVariable,IndexTypeKey,VariableTypeKey, &
                                                     KeyedValue,ReportFreq)
      CHARACTER(len=*), INTENT(IN) :: VariableName    ! String Name of variable
      CHARACTER(len=*), INTENT(IN) :: IndexTypeKey    ! Zone, HeatBalance or HVAC, System, Plant
      CHARACTER(len=*), INTENT(IN) :: VariableTypeKey ! State, Average, or NonState, Sum
      REAL, INTENT(IN), TARGET     :: ActualVariable  ! Actual Variable, used to set up pointer
      CHARACTER(len=*), INTENT(IN) :: KeyedValue      ! Associated Key for this variable
      CHARACTER(len=*), INTENT(IN) :: ReportFreq      ! Frequency to store
'timestep','hourly','monthly','environment'
    END FUNCTION
  END INTERFACE

  INTERFACE GetInternalVariableValue
    REAL FUNCTION GetInternalVariableValue(WhichVar)
      INTEGER, INTENT(IN) :: WhichVar ! Report number assigned to this variable
    END FUNCTION
  END INTERFACE

          ! MODULE VARIABLE DECLARATIONS:

LOGICAL :: BeginDayFlag         ! Set to true at the start of each day, set to false after first
time step in day
LOGICAL :: BeginEnvrnFlag       ! Set to true at the start of each environment, set to false after
first time step in environ
LOGICAL :: BeginHourFlag        ! Set to true at the start of each hour, set to false after first
time step in hour
LOGICAL :: BeginSimFlag         ! Set to true until any actual simulation (full or sizing) has
begun, set to
                                !   false after first time step
LOGICAL :: BeginFullSimFlag     ! Set to true until full simulation has begun, set to
                                !   false after first time step
LOGICAL :: BeginTimeStepFlag    ! Set to true at the start of each time step, set to false after
first subtime step of time step
REAL    :: BigNumber            ! Max Number (real) used for initializations
DOUBLE PRECISION :: DBigNumber  ! Max Number (double precision) used for initializations
INTEGER :: DayOfSim             ! Counter for days (during the simulation)
LOGICAL :: EndEnvrnFlag         ! Set to true at the end of each environment (last time step of
last hour of last day of environ)
LOGICAL :: EndDayFlag           ! Set to true at the end of each day (last time step of last hour
of day)
LOGICAL :: EndHourFlag          ! Set to true at the end of each hour (last time step of hour)
INTEGER :: HourOfDay            ! Counter for hours in a simulation day
INTEGER :: NextHour             ! Next hour index
REAL    :: WeightNextHour       ! Weighting of value for next hour
REAL    :: WeightNow             ! Weighting of value for current hour
INTEGER :: NumOfDayInEnvrn       ! Number of days in the simulation for a particular environment
INTEGER :: NumOfTimeStepInHour   ! Number of time steps in each hour of the simulation
INTEGER :: NumOfZones            ! Total number of Zones for simulation
INTEGER :: TimeStep              ! Counter for time steps (fractional hours)
REAL    :: TimeStepZone          ! Zone time step in fractional hours
LOGICAL :: WarmupFlag            ! Set to true during the warmup portion of a simulation
INTEGER :: OutputFileStandard    ! Unit number for the standard output file (hourly data only)
INTEGER :: OutputFileInits       ! Unit number for the standard Initialization output file
INTEGER :: OutputFileDebug       ! Unit number for debug outputs
INTEGER :: OutputFileZoneSizing  ! Unit number of zone sizing calc output file
INTEGER :: OutputFileSysSizing   ! Unit number of system sizing calc output file
INTEGER :: OutputFileMeters      ! Unit number for meters output
INTEGER :: OutputFileBNDetails   ! Unit number for Branch-Node Details
LOGICAL :: DebugOutput
LOGICAL :: EvenDuringWarmup
LOGICAL :: ZoneSizingCalc = .FALSE.       ! TRUE if zone sizing calculation
LOGICAL :: SysSizingCalc = .FALSE.        ! TRUE if system sizing calculation
LOGICAL :: DoZoneSizing          ! User input in RUN CONTROL object
LOGICAL :: DoSystemSizing        ! User input in RUN CONTROL object
LOGICAL :: DoPlantSizing         ! User input in RUN CONTROL object
LOGICAL :: DoDesDaySim           ! User input in RUN CONTROL object
LOGICAL :: DoWeathSim            ! User input in RUN CONTROL object
LOGICAL :: WeatherFile           ! TRUE if current environment is a weather file
LOGICAL :: DoOutputReporting     ! TRUE if variables to be written out
REAL    :: CurrentTime           ! CurrentTime, in fractional hours, from start of day. Uses Loads
time step.
INTEGER :: SimTimeSteps          ! Number of (Loads) timesteps since beginning of run period
(environment).
INTEGER :: MinutesPerTimeStep    ! Minutes per time step calculated from NumTimeStepInHour (number
of minutes per load time step)
LOGICAL :: DisplayPerfSimulationFlag ! Set to true when "Performing Simulation" should be
displayed


!     NOTICE
!
!     Copyright © 1996-2002 The Board of Trustees of the University of Illinois
!     and The Regents of the University of California through Ernest Orlando Lawrence
!     Berkeley National Laboratory.  All rights reserved.
!
!     Portions of the EnergyPlus software package have been developed and copyrighted
!     by other individuals, companies and institutions.  These portions have been
!     incorporated into the EnergyPlus software package under license.   For a complete
!     list of contributors, see "Notice" located in EnergyPlus.f90.
!
!     NOTICE: The U.S. Government is granted for itself and others acting on its
!     behalf a paid-up, nonexclusive, irrevocable, worldwide license in this data to
!     reproduce, prepare derivative works, and perform publicly and display publicly.
!     Beginning five (5) years after permission to assert copyright is granted,
!     subject to two possible five year renewals, the U.S. Government is granted for
!     itself and others acting on its behalf a paid-up, non-exclusive, irrevocable
!     worldwide license in this data to reproduce, prepare derivative works,
!     distribute copies to the public, perform publicly and display publicly, and to
!     permit others to do so.
!
!     TRADEMARKS: EnergyPlus is a trademark of the US Department of Energy.
!

END MODULE DataGlobals
```

Appendix B. DataEnvironment Module

```
MODULE DataEnvironment      ! EnergyPlus Data-Only Module

          ! MODULE INFORMATION:
          !       AUTHOR         Rick Strand, Dan Fisher, Linda Lawrie
          !       DATE WRITTEN   December 1997
          !       MODIFIED       November 1998, Fred Winkelmann
          !       MODIFIED       June 1999,June 2000, Linda Lawrie
          !       RE-ENGINEERED  na

          ! PURPOSE OF THIS MODULE:
          ! This data-only module is a repository for the variables that relate specifically
          ! to the "environment" (i.e. current date data, tomorrow's date data, and
          ! current weather variables)

          ! METHODOLOGY EMPLOYED:
          ! na

          ! REFERENCES:
          ! na

          ! OTHER NOTES:
          ! na

          ! USE STATEMENTS:
USE DataGlobals, ONLY: MaxNameLength
          ! None!--This module is USEd by all other modules; it should not USE anything.

IMPLICIT NONE   ! Enforce explicit typing of all variables

PUBLIC          ! By definition, all variables which are placed in this data
                ! -only module should be available to other modules and routines.
                ! Thus, all variables in this module must be PUBLIC.


          ! MODULE PARAMETER DEFINITIONS:
          ! na

          ! DERIVED TYPE DEFINITIONS
          ! na

          ! INTERFACE BLOCK SPECIFICATIONS
          ! na

          ! MODULE VARIABLE DECLARATIONS:

REAL    :: BeamSolarRad            ! Current beam normal solar irradiance
REAL    :: CosSolarDeclinAngle     ! Cosine of the solar declination angle
INTEGER :: DayOfMonth              ! Current day of the month
INTEGER :: DayOfMonthTomorrow      ! Tomorrow's day of the month
INTEGER :: DayOfWeek               ! Current day of the week (Sunday=1, Monday=2, ...)
INTEGER :: DayOfWeekTomorrow       ! Tomorrow's day of the week (Sunday=1, Monday=2, ...)
INTEGER :: DayOfYear               ! Current day of the year (01JAN=1, 02JAN=2, ...)
REAL    :: DifSolarRad             ! Current sky diffuse solar horizontal irradiance
INTEGER :: DSTIndicator            ! Daylight Savings Time Indicator (1=yes, 0=no) for Today
REAL    :: Elevation               ! Elevation of this building site
LOGICAL :: EndMonthFlag            ! Set to true on last day of month
REAL    :: EquationOfTime          ! Value of the equation of time formula
REAL    :: GndReflectanceForDayltg ! Ground visible reflectance for use in daylighting calc
REAL    :: GndReflectance          ! Ground visible reflectance from input
REAL    :: GndSolarRad             ! Current ground reflected radiation
REAL    :: GroundTemp              ! Current ground temperature
INTEGER :: HolidayIndex            ! Indicates whether current day is a holiday and if so what
type
                                   ! HolidayIndex=(0-no holiday, 1-holiday type 1, ...)
INTEGER :: HolidayIndexTomorrow    ! Tomorrow's Holiday Index
LOGICAL :: IsRain                  ! Surfaces are wet for this time interval
LOGICAL :: IsSnow                   ! Snow on the ground for this time interval
REAL    :: Latitude                 ! Latitude of building location
REAL    :: SinLatitude              ! Sine of Latitude
REAL    :: CosLatitude              ! Cosine of Latitude
REAL    :: Longitude                ! Longitude of building location
INTEGER :: Month                    ! Current calendar month
INTEGER :: MonthTomorrow            ! Tomorrow's calendar month
REAL    :: OutBaroPress             ! Current outdoor air barometric pressure
REAL    :: OutDryBulbTemp           ! Current outdoor air dry bulb temperature
REAL    :: OutHumRat                ! Current outdoor air humidity ratio
REAL    :: OutRelHum                ! Current outdoor relative humidity [%]
REAL    :: OutRelHumValue           ! Current outdoor relative humidity value [0.0-1.0]
REAL    :: OutEnthalpy              ! Current outdoor enthalpy
REAL    :: OutAirDensity            ! Current outdoor air density
REAL    :: OutWetBulbTemp           ! Current outdoor air wet bulb temperature
REAL    :: OutDewPointTemp          ! Current outdoor dewpoint temperature
REAL    :: SinSolarDeclinAngle      ! Sine of the solar declination angle
REAL    :: SkyTemp                  ! Current sky temperature
LOGICAL :: SunIsUp                  ! True when Sun is over horizon, False when not
REAL    :: WindDir                  ! Current outdoor air wind direction
REAL    :: WindSpeed                ! Current outdoor air wind speed
INTEGER :: Year                     ! Current calendar year of the simulation
INTEGER :: YearTomorrow             ! Tomorrow's calendar year of the simulation
REAL, DIMENSION(3)  :: SOLCOS       ! Solar direction cosines at current time step
REAL    :: CloudFraction            ! Fraction of sky covered by clouds
REAL    :: HISKF                    ! Exterior horizontal illuminance from sky (lum/m^2).
REAL    :: HISUNF                   ! Exterior horizontal beam illuminance (lum/m^2)
REAL    :: SkyClearness             ! Sky clearness (see subr. DayltgLuminousEfficacy)
REAL    :: SkyBrightness            ! Sky brightness (see subr. DayltgLuminousEfficacy)
REAL    :: StdBaroPress             ! Standard "atmospheric pressure" based on elevation (ASHRAE
HOF p6.1)
REAL    :: TimeZoneNumber           ! Time Zone Number of building location
REAL    :: TimeZoneMeridian         ! Standard Meridian of TimeZone
CHARACTER(len=MaxNameLength) :: EnvironmentName ! Current environment name
CHARACTER(len=20)  :: CurMnDyHr     ! Current Month/Day/Hour timestamp info
CHARACTER(len=5)   :: CurMnDy       ! Current Month/Day timestamp info
INTEGER :: CurEnvirNum              ! current environment number
Integer :: TotDesDays               ! Total number of Design days to Setup

!     NOTICE
!
!     Copyright © 1996-2002 The Board of Trustees of the University of Illinois
!     and The Regents of the University of California through Ernest Orlando Lawrence
!     Berkeley National Laboratory.  All rights reserved.
!
!     Portions of the EnergyPlus software package have been developed and copyrighted
!     by other individuals, companies and institutions.  These portions have been
!     incorporated into the EnergyPlus software package under license.   For a complete
!     list of contributors, see "Notice" located in EnergyPlus.f90.
!
!     NOTICE: The U.S. Government is granted for itself and others acting on its
!     behalf a paid-up, nonexclusive, irrevocable, worldwide license in this data to
!     reproduce, prepare derivative works, and perform publicly and display publicly.
!     Beginning five (5) years after permission to assert copyright is granted,
!     subject to two possible five year renewals, the U.S. Government is granted for
!     itself and others acting on its behalf a paid-up, non-exclusive, irrevocable
!     worldwide license in this data to reproduce, prepare derivative works,
!     distribute copies to the public, perform publicly and display publicly, and to
!     permit others to do so.
!
!     TRADEMARKS: EnergyPlus is a trademark of the US Department of Energy.
!

END MODULE DataEnvironment
```